MASTER THESIS

zur Erlangung des akademischen Grades
„Master of Science in Engineering“
im Studiengang Softwareentwicklung

Erstellung eines Prozesses für die Planungsphase der Softwarearchitektur

Ausgeführt von: Bernhard Posselt, BSc
Personenkennzahl: 1310299032

Begutachterin: Mag. Maria-Therese Teichmann

Wien, den 7. Mai 2015

FACHHOCHSCHULE
TECHNIKUM WIEN

Eidesstattliche Erklärung

„Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Ich versichere, dass die abgegebene Version jener im Uploadtool entspricht."

Wien, 7. Mai 2015 Unterschrift

Kurzfassung

Softwarearchitektur ist ein sehr breites Gebiet und sehr abstraktes Gebiet der Softwareentwicklung, ist aber Voraussetzung für qualitativ hochwertige Software. Der hohe Abstraktionsgrad ist aber oft ein Hindernis für eine dem/der KundIn kommunizierbare, reproduzierbare und qualitativ hochwertige Architektur.

Deswegen wird in dieser Arbeit ein reproduzierbarer und kommunizierbarer Architekturerstellungsprozess entworfen. Da laut Zehner-Regel der Fehlerkosten die Behebungskosten unentdeckter Fehler im Laufe des Projektes exponentiell ansteigen, beschränkt sich der Prozess auf die frühe Architekturplanungsphase, um eine möglichst große Wirkung zu erzielen.

Nach mehreren Versuchen, welche wie viele Architekturreviewmethoden stark auf die Priorisierung der Qualitätsmerkmale der Anforderungen aufbauten, wurde klar, dass sich in der frühen Planungsphase aufgrund der fehlenden Implementation zu wenig messbare Werte für einen reproduzierbaren Prozess ermitteln lassen.

Der Prozess baut daher auf verfügbaren Werten wie Nachbarsystemen, Netzwerken und Risikokosten im Bezug auf unerlaubten Datenzugriff auf. Zusätzlich versucht er durch Analysen der ermittelten Komponentenstruktur Hinweise auf in der Implementationsphase problematische Bereiche zu geben.

Der schließlich erstellte Prozess ist reproduzierbar und die resultierende Architektur aufgrund der Kosten und verwendeten Modellierungssprache UML gut dem/der KundIn kommunizierbar.

Schlagworte: Softwarearchitektur, Softwareentwicklung, Qualität

Abstract

Software architecture is a very wide and abstract subject in the field of software development, but is required to create high-quality software. Because of the high degree of abstraction, communicating and creating a reproducible, high-quality software architecture is often very hard.

Therefore this thesis will outline a reproducible and communicable software architecture process. As errors are more costly to fix in a later phase of software development, the process itself focuses only on the planning phase in order to achieve high levels of efficiency.

After various attempts to create a software architecture process based on prioritized quality factors which in turn is also often used in many software review processes, it became obvious that there were too few measurable values to create a reproducible process. This was mostly a result of the missing implementation.

The process therefore builds on early available information such as adjacent systems, networks and costs which would be created through unauthorized data access. Additionally, the process analyses the component structure to reveal possible risk areas which should be closely monitored in the following implementation phase.

The result is a reproducible piece which, due to its cost-based approach and modeling language UML, can be communicated well to the client.

Keywords: software architecture, software development, quality

Inhaltsverzeichnis

1 Einführung

Softwarearchitektur ist ein noch ein sehr junges Feld in der Softwarebranche und oft schwer greif- und kommunizierbar [33, S. 8], ist jedoch für die Qualität der erstellten Software unabdingbar. Je mehr Unternehmen ihre Datenverwaltung durch Softwarelösungen umsetzen, desto offensichtlicher werden die durch mangelhafte Architekturen verursachten Probleme. Oliver Vogel zählt unter Anderem folgende Symptome auf, welche primär auf schlechte Architekturentscheidungen zurückzuführen sind [33, S. 7]:

- „Schnittstellen, die schwer zu verwenden bzw. zu pflegen sind weil sie einen zu großen Umfang haben“
- „Quelltext, der an zahlreichen Stellen im System angepasst werden muss, wenn Systembausteine, wie beispielsweise Datenbank oder Betriebssystem, geändert werden“
- „Klassen, die sehr viele ganz unterschiedliche Verantwortlichkeiten abdecken und deshalb schwer wiederzuverwenden sind“
- „Fachklassen, deren Implementierungsdetails im gesamten System bekannt sind“

Diese Probleme treten jedoch nicht nur auf der Sourcecodeebene auf: Nimmt die Größe eines Systems zu, können sich die Ressourcen einer einzelnen Komponente als unzureichend erweisen; das System muss folglich vertikal skaliert werden, sprich die Leistung muss durch das Hinzufügen weiterer Komponenten erhöht werden.

Ein weiterer Grund für die Aufspaltung eines Systems in mehrere Komponenten ist die inhärente Komplexität der einzelnen Systeme, welche Fehler und Sicherheitslücken begünstigt. Sind zB. alle wichtigen Daten auf einem System gespeichert, so können Fehler, welche auf diesem System auftreten, einen Komplettausfall des Systems verursachen. Verwaltet das Unternehmen auch oft verwendete Geschäftsdaten auf diesem System, so verursacht der Ausfall des Systems einen Arbeitsstillstand: Daten können weder eingegeben noch abgerufen werden und verursachen dem Unternehmen dadurch signifikante Kosten.

Das gleiche Problem kann auch durch die böswillige Ausnützung von Sicherheitslücken auftreten: Hat ein/eine AngreiferIn erst einmal die Kontrolle über ein System, kann er auf sämtliche auf dem System gespeicherten Daten zugreifen, diese manipulieren und das System in einen unbenutzbaren Zustand versetzen. Dass dies kein unrealistisches Szenario darstellt, wird durch die in letzter Zeit veröffentlichten Sicherheitslücken wie Heartbleed[3] und die Remote Execution Lücke im IIS ersichtlich[22], welche AngreiferInnen die Möglichkeit zur vollen Kontrolle des Systems gibt. Eine Lösung kann auch hier die Aufspaltung in mehrere Systeme darstellen, um wichtige Daten durch eine Verringerung der Angriffsfläche zu schützen.

Eine gut ausgelegte Komponentenarchitektur ist somit nicht nur für große Systeme wichtig, um eine effiziente Verarbeitung von Daten zu garantieren, sondern auch für mittlere Systeme, welche zB. bei mittelständischen Unternehmen eingesetzt werden.

1.1 Forschungsleitende Fragen

Die Grundfragen, welche diese Arbeit beantworten soll sind folgende:

- Wie kommt man von Anforderungen auf eine gute Architektur?
- Gibt es einen allgemein gültigen, anwendbaren Architekturprozess, welcher es erlaubt, reproduzierbar Komponenten zu erstellen?

Die erste Grundfrage, welche die Wahl dieses Themas wesentlich beeinflusst hat, ist die Frage: Wie kommt man von Anforderungen auf eine gute Architektur. Anders formuliert: Wie kann eine qualitativ hochwertige Softwarearchitektur mit den verfügbaren Anforderungen erstellt werden kann.

Die zweite Grundfrage baut direkt auf der Ersten auf und dreht sich um die Erstellung von Komponenten durch einen definierten Prozess. Was genau ist jedoch ein Prozess? Das Merriam-Webster Lexikon definiert einen Prozess als eine Abfolge von Handlungen, die entweder etwas produzieren oder zu einem bestimmten Resultat führen[21]. Diese Handlungen können nicht nur qualitätssichernde Schritte beinhaltet, um einen gewissen Qualitätsstandard bei der Architekturerstellung zu garantieren, sondern auch eine Effizienzsteigerung ermöglichen.

Der Prozess soll allgemein anwendbar und reproduzierbar sein, kann also nicht wie die meisten Architekturprozessbeschreibungen auf einer zu abstrakten Ebene definiert werden.

1.2 Methodik

Um die Fragestellungen zu untersuchen und zu beantworten, wird anhand eines Beispielprojekts, Anforderungen und Architekturreviews versucht, eine Architektur zu erstellen. Die Probleme bei der Erstellung sollen dokumentiert werden und aus der Architektur soll ein allgemein anwendbarer Prozess ermittelt werden.

Das Projekt dreht sich um ein System einer Personenzertifizierungsstelle. Die Zertifizierungsstelle selbst ist ein mittelständischer, durchschnittlicher Betrieb. Das System der Zertifzierungsstelle muss die Vorgaben des ISO Standards für Personenzertifizierungsstellen erfüllen[12] und stellt diverse Verwaltungsfunktionalitäten bereit. Dazu zählen zB. die Verwaltung und Auswertung von Zertifizierungssprüfungen, das Erstellen von Prüfungsterminen, für welche sich Personen anmelden können und die Verwaltung des Zahlungsverkehrs.

Durch die Rahmenbedinungen des einzuhaltenden ISO Standards stehen vor allem Datenschutz und Sicherheit im Vordergrund. Der Hohe Fokus auf Datenschutz und Sicherheit mag

zwar eine Eigenheit des Projekts sein, ist aber durchaus relevant und attraktiv für Projekte in anderen Bereichen.

Regelungen bezüglich Datenschutz sind zB. fest im österreichischen Gesetz für Datenschutz verankert. Das Gesetz schreibt vor, dass „die Zugriffsberechtigung auf Daten und Programme und der Schutz der Datenträger vor der Einsicht und Verwendung durch Unbefugte“[1, § 14, 5] geregelt werden müssen, sowie „die Berechtigung zum Betrieb der Datenverarbeitungsgeräte“ festgelegt „und jedes Gerät durch Vorkehrungen bei den eingesetzten Maschinen oder Programmen gegen die unbefugte Inbetriebnahme“[1, § 14, 6] abgesichert werden muss. Somit ist grundsätzlich jedes Softwareprojekt, welches in Österreich eingesetzt wird und personenbezogene Daten verwaltet zur Einhaltung dieser gesetzlichen Rahmenbedingungen verpflichtet.

Auch freiwillige Zertifizierungen im Bezug auf Datenschutz und Sicherheit, wie zB. wie ISO 27001 [13], können sich für Unternehmen rechnen. So kann zB. mit Gütesiegeln und Zertifzierungen geworben und somit der höhere Preis gegenüber konkurrierenden Firmen gerechtfertigt werden. Dies ist besonders relevant, da zB. konkurrierenden Firmen aus dem Ausland durch niedrigere Löhne und bessere Voraussetzungen Angebote österreichischer Firmen weit unterbieten können.

Aufgrund der Vermutung, dass der Anforderungsprozess eine große Rolle für den Architekturprozess spielt, sollen zusätzliche, architekturrelevante Anforderungsparameter ermittelt und für weitere Analysen und Entscheidungen im Prozess verwendet werden.

Da Architekturreviews wie ATAM stark auf nicht funktionale Anforderungen setzen und die Architektur selbst diese stark beeinflusst, soll versucht werden, diese als Hauptentscheidungsgrundlage für die Architekturerstellung zu verwenden. Diese Entscheidungen sollen nach einem festgelegten Regelwerk getroffen werden, um die Varianz, welche Architekturreviews wie ATAM und CBAM inne wohnen, zu reduzieren. Dies soll eine Mindestqualität der Architektur nach der Durchführung des Prozesses garantieren.

Die Architektur, welche durch den Prozess erstellt werden soll, soll kein zu kompliziertes und großes System werden. Deswegen soll die Erstellung jeder Komponente durch eine Anforderung gerechtfertigt werden können.

Um die Verständlichkeit zu erhöhen sollen die Artefakte des Prozesses, falls möglich, durch UML dargestellt werden. Der Hauptfokus soll auf der Erstellung der Komponenten und Interfaces liegen.

Die Implementationsphase wird zwar nicht mehr behandelt, für sie sollen jedoch verwertbare Ausgangsparameter gefunden werden, welche Risiken aufdecken und Entscheidungshilfen für die Implementation bereit stellen.

1.3 Abgrenzung

Eine Abdeckung des kompletten Architekturprozesses ist aufgrund der Tiefe und des Umfangs nicht möglich. Würde dies versucht werden, müsste dies zudem auf einem höherer Abstraktionsgrad geschehen, was wiederum die Anwendbarkeit des Prozesses erschwert. Der Pro-

zess selbst beschränkt sich deswegen auf die Planungsphase der Architekturerstellung. Der Anforderungs- und Implementationsprozess werden als umschließende Abschnitte zwar mit einbezogen, jedoch liegt der Fokus nur auf den Eingangs- respektive Ausgangsparametern.

Eine gute Verständlichkeit und Kommunizierbarkeit des Prozesses gehören zwar zu den erklärten Zielen, jedoch werden für diese Bereiche keine eigenen Richtlinien und Vorgaben erstellt. Diese Bereiche überschneiden sich stark mit den Feldern der Kommunikationswissenschaft und Psychologie, welche nicht im Fokus dieser Arbeit stehen. Auch der Anforderungsprozess überschneidet sich wesentlich mit diesen Bereichen und wird daher nicht ganzheitlich beschrieben: Er verwendet zB. bestimmte Frage- und Dokumentationstechniken, mit welchen versucht werden soll, Anforderungen komplett, korrekt und verständlich zu erfassen. Da der Anforderungsprozess aber die Eingabeparameter und Ziele für den Architekturprozess liefert, ist dieser für den Architekturprozess wichtig. Die Arbeit beschränkt sich deswegen auf die Beschreibung architekturrelevanter Ausgangsparameter, welche im Anforderungsprozess ermittelt werden.

Das Beispielprojekt selbst soll nur in der Architekturplanungsphase durchgeführt werden. Die Anzahl und Qualität der Anforderungen liefert genügend Parameter für die Erprobung des Prozesses, die Implementation des Systems selbst ist jedoch aufgrund der Anforderungen zu zeit- und aufwandsintensiv: Nicht nur die zu implementierenden Funktionen sind hier zu zahlreich, sondern auch die Maßnahmen, welche für die benötigten Qualitäten - zB. Datenschutz - durchgeführt werden müssten.

Durch die fehlende Implementation wird daher auch auf die genaue Planung der Codestruktur verzichtet. Das bedeutet nicht nur, dass auf Patterns verzichtet wird, sondern auch, dass Architektursichten, welche sich mit der Planung des Codes beschäftigen nicht beachtet werden; dazu zählt zB. Kruchtens Development View. Grundsätzlich wäre die Planung der Module zwar möglich, aber wegen der fehlenden Implementation nicht auf ihre Korrekt- und Angemessenheit überprüfbar.

Da Architektur stark von den Anforderungen an das Systems abhängig ist, kann durch zu dominante Anforderungen ein komplett anderes Vorgehen benötigt werden. Dies trifft nicht nur auf Gebiete wie Embedded zu, wo durch die fehlende Rechenkraft und geringen Speicherplatz so effektiv wie möglich gearbeitet werden muss, sondern auch auf Systeme, welche durch ihre Größe, Datenmengen und Rechenleistungen eine eigene Aufteilung und Planung benötigen.

Auch Rahmenbedingungen haben einen großen Einfluss auf die Architektur: Soll zB. ein System in einer sehr kurzen Zeit erstellt werden oder ist eine bestimmte Technologie wie CORBA vorgeschrieben, kann dies auch die Wahl der Komponenten beeinflussen. All diese Ausnahmefälle sind hinderlich für die Erstellung eines simplen und verständlichen Prozesses. Deswegen wird bei der Erstellung auf einen kompletten, auf jede Ausnahme anwendbaren Prozess verzichtet.

Die in der Arbeit verwendeten Architekturreviewmethoden werden auf ATAM und CBAM begrenzt. ATAM ist die ausgereifteste Reviewmethode [26, S. 184], auf welcher viele andere Reviewmethoden wie ALMA und CBAM aufbauen. CBAM ist wegen des Preis-Leistungsansatzes

interessant für den Prozess.

Die nach der initialen Architekturerstellung durchgeführten Analysen beanspruchen keine hundertprozentige Abdeckung aller Möglichkeiten. Auch die durchgeführten Berechnungen sind nicht bis ins kleinste Detail genau und können je nach Projekt in ihrer Genauigkeit variieren. Sie sollen vor allem auf die Analysemöglichkeiten hinweisen, welche schon vor einem Architekturreview durchführbar sind.

1.4 Übersicht

Die Arbeit teilt sich in folgende Kapitel auf:

- Softwarequalität
- Softwarearchitektur
- Modellierung in der Architektur
- Prozesserstellungsversuche
- Ermittlung der Architekturanforderungen
- Erstellung der Architektur
- Zusammenfassung

1.4.1 Softwarequalität

Die Sicherstellung der Softwarequalität ist eines der wichtigsten Ziele bei der Erstellung der Softwarearchitektur, unter Anderem weil die Architektur die Struktur der zu erstellenden Software mitbestimmt und somit gewisse Qualitätsmerkmale begünstigt oder limitiert.

Um dieses Ziel zu verstehen, muss zuerst definiert werden, was Softwarequalität respektive Qualität überhaupt bedeutet. Eine Antwort darauf liefert der in ISO 9126 beschriebene Standard und dessen Nachfolger ISO 25010, auf welchen aber aus geringen Verbreitungsgründen nur kurz hingewiesen wird.

ISO 9126 definiert Softwarequalität in der Erfüllung der Anforderungen. Diese lassen sich in funktionale und nicht funktionale Anforderungen aufspalten, wobei bei der Architekturerstellung der primäre Fokus auf den nicht funktionalen Anforderungen liegt.

1.4.2 Softwarearchitektur

Architektur versucht durch Abstraktion eine Übersicht über das zu erstellende System zu gewähren und legt die Grundpfeiler der Software fest. Dies wird standardmäßig durch die Verwendung von Architektursichten erreicht. Es gibt mehrere verschiedene Modelle, welche unterschiedliche Sichten definieren. Um zu erläutern, wie diese Sichten genau verwendet werden

können, kann unter Anderem das Modell von Kruchten und das Zachman Framework herangezogen werden.

Wie gestaltet sich jedoch der genaue Ablauf der Architekturerstellung? Der Architekturprozess kann in mehrere Abschnitte aufgeteilt werden: Nach einem kurzen Anfangsprozess nach der Anforderungsanalyse kann mit der Planung, Erstellung, Überprüfung der Architektur begonnen werden. Dabei muss auch die Kommunikation beachtet werden.

Für die Überprüfung der Architektur werden Architekturreviewframeworks verwendet. Das Bekannteste unter ihnen ist ATAM, welches durch mehrere Szenarientypen die Qualitätsmerkmale der gewählten Architektur zu überprüfen versucht. Eine weitere Reviewmethode ist CBAM, welche auf ATAM aufbaut aber vor Allem das Preis-Leistungsverhältnis als Entscheidungsgrundlage verwendet.

1.4.3 Modellierung in der Architektur

Modelle besitzen ein verkürzende Wirkung und bieten sich damit vor Allem zur Dokumentation und Kommunikation der Architektur und Anforderungen an. Die in ISO 42010 beschriebenen Architektursichten verlangen zudem ein oder mehrere Architekturmodelle. Diese Modelle können mit Hilfe von auf die Architektur zugeschnittenen Modellierungssprachen umgesetzt werden.

Hier bietet wegen der großen Verbreitung vor allem UML an. UML bietet alle für die Architektur nötigen Modelle. Es werden jedoch nicht alle Modelle benötigt. Für die Arbeit werden lediglich das Usecase-, Komponenten-, Klassen- und Aktivitätsdiagramm verwendet. Zusätzlich wird ein Kontextdiagramm eingesetzt, welches jedoch nicht Teil von UML ist.

1.4.4 Prozesserstellungsversuche

Anhand der priorisierten nicht funktionale Anforderungen wurde nun versucht, einen Architekturprozess abzuleiten. Dies scheiterte jedoch aufgrund der Anforderung an den Prozess: Der Prozess soll allgemein anwendbar und reproduzierbar sein; die Bewertung der nicht funktionalen Anforderungen war aber in der Planungsphase durch die fehlende Implementation zu ungenau. Auch durch die Verwendung von Kohäsionswerten ließ sich keine eindeutige Architekturerstellung ableiten und die Architekturen wurden zu teuer.

Erst die Aufspaltung in Daten und Zugriffe durch AkteurInnen verhalf zu einer reproduzierbaren Architektur, welche groß genug für Bewertungen war und nicht zu teuer war.

1.4.5 Ermittlung der Architekturanforderungen

Der Architekturprozess basiert auf Artefakten aus dem Anforderungsprozess. Je früher Anforderungen bekannt sind, desto früher können diese in den Architekturprozess einbezogen werden, um Fehlerfolgekosten zu sparen. Dazu zählen besonders architekturrelevante Anforderungen. Außerdem können bereits früh Szenarien für die Verwendung in Architketurreviews

ermittelt werden. Diese Szenarien können auch in einer frühen Architekturanalyse verwendet werden.

1.4.6 Erstellung der Architektur

Auf Basis der in der Anforderungsphase ermittelten Anforderungen und Szenarien kann nun begonnen werden, die Komponenten der Architektur zu erstellen. Jedes Netz wird durch Gateways getrennt, welche nur erlaubte Anfragen durch lassen. Zusätzlich wird eine Angriffskosten der involvierten AkteurInnen durchgeführt, um zu ermitteln, ob eine weitere Aufteilung von Nöten ist.

Sind alle Komponenten erstellt, kann mit einer Analyse der nicht funktionalen Anforderungen begonnen werden. Diese Analysen liefern grobe Ergebnisse, dienen aber schon als Hinweis für die Implementierungsphase und zeigen mögliche Probleme und Verbesserungsbereiche auf.

2 Softwarequalität

Für viele KundInnen ist Softwarearchitektur ein schwer einorden- und erklärbares Teilgebiet der Softwareentwicklung. Deswegen ist es schwierig, den/die KundIn für die Mithilfe an der Architekturerstellung und die daraus resultierenden Kosten zu gewinnen. Eine mögliche Lösung für dieses Problem ist es, die durch unzureichende Softwarequalität entstandenen Kosten, welche wiederum die Folge einer mangelhaften Architektur sind, den Kosten der Architekturerstellung gegenüber zu stellen. [33, S. 8-9]

Eine gute Softwarearchitektur allein kann zwar nicht eine hohe Softwarequalität garantieren, ist jedoch ein wichtiger Faktor, ohne welchen keine hochqualitative Software erstellt werden kann. Softwarequalität und Softwarearchitektur stehen somit in einem engen Zusammenhang. [33, S. 8-9]

Was genau verbirgt sich jedoch hinter dem Begriff Softwarequalität? Der Begriff Qualität selbst ist sehr breit und schwer definierbar. Laut Garvin gibt es fünf verschiedene Herangehensweisen [4, S. 25-29]:

- Transzendentale Sicht
- Produktbasierte Sicht
- Userbasierte Sicht
- Herstellungsbasierte Sicht
- Wertbasierte Sicht

Die transzendentale Sicht beschreibt Qualität als sofort erkennbar, aber als nicht wirklich beschreibbar. Die produktbasierte Sicht ordnet die Qualität des Produktes dessen Bestandteilen zu. Bei der userbasierten Sicht geht es darum, die Anforderungen des/der NutzerInnen zu befriedigen: Die höchste Qualität eines Produktes wird daher erreicht, wenn der/die BenutzerIn alles vorfindet, was er/sie sich erwartet, nicht Mehr und nicht Weniger. Die herstellungsbasierte Sicht misst Qualität an der Erfüllung der Spezifikation und des Designs. Die wertbasierte Sicht schließlich definiert die Qualität durch das Preis-Leistungsverhältnis. [4, S. 26]

Diese Sichten treffen mehr oder weniger auch für Softwarequalität zu [27, S. 399], sind aber noch zu grob formuliert, um konkrete Schritte bei der Erstellung des Systems und dessen Architektur zu setzen und konkrete Werte für die Überprüfung der Qualität fest zu legen. Eine genaue Auflistung der für Softwarequalität relevanten Faktoren, bzw. ein Modell, welches den Qualitätsprozess unterstützt, ist somit von Nöten.

2.1 Softwarequalitätsmodelle

ISO 9126 [9] und dessen Nachfolger ISO 25010 [11] bieten ein Modell, um Softwarequalität zu beschreiben. ISO 25010, der Nachfolgestandard, erweitert die in ISO 9126 beschriebenen Hauptkategorien um Security und Compatibility. Auch die Kategorie Software Quality in Use erfährt eine Überarbeitung: Sie enthält nun die Kategorie Usability, welche vorher in den Hauptkriterien definiert war. Da es aber schwierig war, zitierbare Quellen zum neuen Standard zu finden - ISO 25010 hat „in die Praxis wenig Einzug gehalten“ [32, S. 60] - , die Neuerungen überschaubar und mehr eine Umorganisierung als Revolution darstellen, wird auf die Nutzung von ISO 25010 verzichtet und das Qualitätsmodell des Vorgängers, ISO 9216, verwendet.

ISO 9126 definiert folgende Qualitätskategorien:

- „Functionality“
- „Reliability“
- „Usability“
- „Efficiency“
- „Maintainability“
- „Portability“

Die beschriebenen Kategorien können laut dem International Requirements Engineering Board, kurz IREB, wiederum in folgende Überkategorien eingeteilt werden: funktionale Anforderungen und nicht funktionale Anforderungen (Abbildung 1).

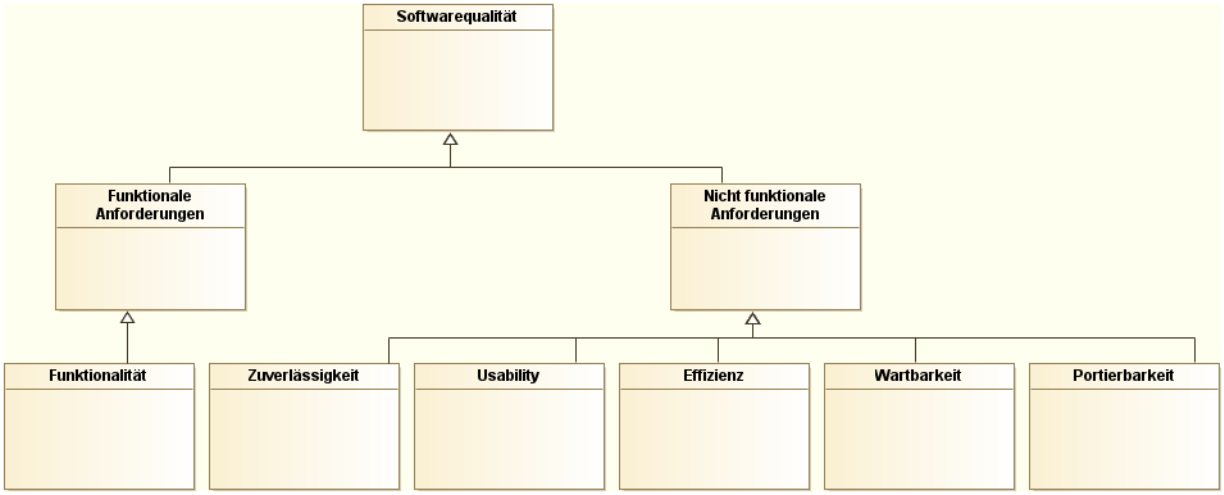

Abbildung 1: Die Qualitätskategorien des ISO 9126 Standards unterteilt in funktionale und nicht funktionale Anforderungen

2.2 Funktionale Anforderungen

Funktionale Anforderungen beschreiben die Funktionalität des Systems, welche die Geschäftsprozesse des/der KundIn umsetzen und das System für ihn/sie somit wertvoll machen [19, S. 79]. Sie werden vom/von der KundIn meist in der Form von Usecases beschrieben [19, S. 78]

IREB ordnet die Functionality Kategorie von ISO 9126 den funktionalen Anforderungen zu[7, S. 9], welche sich in folgende Unterkategorien aufspaltet:

- „Suitability“
- „Accuracy“
- „Interoperability“
- „Security“
- „Functional compliance“

Suitability beschreibt das Vorhandensein von Funktionen, welche die vom/von der NutzerIn geforderten Funktionalitäten bereitstellen. Accuracy behandelt die geforderte Genauigkeit der implementierten Funktionen, Interoperability wiederum die Interaktionsmöglichkeiten des Systems mit anderen Systemen. Security beschreibt die implementierten Kontroll- und Sicherheitsmechanismen, mit welchen das System die Daten vor unerlaubtem Zugriff schützt. Der letzte Punkt, Functional Compliance, beschreibt die Standard- und Gesetzeskonformität des Softwareprodukts. [9, S. 8]

2.3 Nicht funktionale Anforderungen

Nicht funktionalen Anforderungen „verkörpern Erwartungen und Notwendigkeiten, die von InteressensvertreterInnen (AuftraggeberIn, BenutzerIn, ArchitektIn, EntwicklerIn, etc.) neben den funktionalen Anforderungen als wichtig erachtet werden und über die reine gewünschte Funktionalität hinausgehen“[33, S. 108]. Sie definieren nicht das Was sondern das Wie [19, S. 80].

Der Fokus liegt meist auf der Funktionalität eines Systems; es ist schließlich der Hauptgrund für die Erstellung des Systems und der/die KundIn hat eine genaue Vorstellung, was das System genau können muss. Die Qualität des Systems ist meist eine implizite Anforderung, welche im Anforderungsprozess besonders beachtet werden muss, aber trotzdem essentiell für den Erfolg des Systems ist. [33, S. 109]

Dies lässt an folgendem Beispiel demonstrieren: Ein/Eine KundIn beauftragt eine Firma, einen Webshop zu erstellen, auf welchem er/sie seine/ihre Produkte verkaufen will. Die Funktionalität des Webshops wird wie beschrieben implementiert, aber das Endprodukt ist so langsam, dass ein Großteil der KundInnen den Bestellvorgang abbricht. Der Hauptfunktion des Systems, nämlich Produkte verkaufen, ist somit zwar prinzipiell möglich, aber für den/die KundIn im Vergleich zur investierten Summe unrentabel.

Nicht funktionale Anforderungen sind nicht nur oft implizite Anforderungen, sondern auch schwer bezifferbar: Oft wird zB. einfach verlangt, dass das System schnell sein soll. Eine genaue Definition, was der/die KundIn unter schnell versteht ist schwer zu ermitteln. Außerdem sind durch die Kontextabhängigkeit des Begriffes keine allgemeingültigen Werte ermittelbar: Eine ein Sekunden lange Antwortzeit kann für die Nutzung einer Buchhaltungssoftware schnell genug sein, werden die Daten aber für zeitkritische Anwendungen wie Aktienkäufe benötigt, ist die selbe Antwortzeit inakzeptabel. [32, S. 59].

Es sind jedoch besonders die nicht funktionalen Anforderungen, welche im Fokus der Architekturerstellung liegen: Durch die in der Architekturphase entwickelte Struktur der Applikation wird die Erfüllung bestimmter Qualitätsmerkmale überhaupt erst möglich. Die Architektur beinflusst somit die nicht funktionalen Qualitäten eines Systems stark [33, S. 109]. Dies erlaubt folgenden Umkehrschluss: „Architektur ist für die Qualität eines Systems notwendig“[32, S. 59]. [32, S. 19]

Weil die Qualität des Systems wie beschrieben maßgeblich von der Architektur abhängt, welche wiederum von den Qualitätsmerkmalen nicht funktionaler Anforderungen abhängt, ist auch die Ermittlung, Korrektheit und Präzision dieser Anforderung für die Architektur von hoher Bedeutung.

Die nicht funktionalen Anforderungen in ISO 9126 werden vom IREB den verbleibenden Kategorien zugeschrieben. Darunter fallen folgende Kategorien[7, S. 9]:

- „Suitability“
- „Accuracy“
- „Interoperability“

- „Security“
- „Functional compliance“

All diese Kategorien haben wie die Functionality Kategorie eigene Unterpunkte.

2.3.1 Reliability

Die Reliability Kategorie beschreibt die Zuverlässigkeit des Systems unter Last und besteht aus folgenden Unterpunkten [9, S. 7]:

- „Maturity“
- „Fault tolerance“
- „Recoverability“
- „Reliability compliance“

Maturity beschreibt die Fähigkeit des Systems, Ausfälle des Systems wegen Softwarefehlern zu vermeiden; Fault Tolerance wiederum behandelt die Fähigkeit des Systems eine gewisse Leistung des Systems trotz Softwarefehler zu ermöglichen. Recoverability beschreibt Fähigkeiten eines Systems sich von Fehlern zu erholen. Reliability Compliance bezieht sich auf die Konformität gegenüber Standards und Konventionen. [9, S. 8-9]

2.3.2 Usability

Die Usability Kategorie beschreibt die Benutzbarkeit des Systems und besteht aus folgenden Unterpunkten [9, S. 7]:

- „Understandability“
- „Learnability“
- „Operability“
- „Attractiveness“
- „Usability compliance“

Understandability beschreibt die Einfachheit, mit welcher ein/eine BenutzerIn das System für eine bestimmte Aufgabe als verwendbar klassifiziert, Learnability beschreibt, wie leicht ein/eine BenutzerIn das System erlernen kann. Operability beschreibt, in wieweit ein/eine BenutzerIn das System bedienen und kontrollieren kann. Actractiveness beschreibt, ob das Aussehen von den NutzerInnen als attraktiv wahrgenommen wird. Usability Compliance schließlich beschreibt die Konformität gegenüber Standards, Styleguides und Konventionen. [9, S. 9-10]

2.3.3 Efficiency

Die Efficiency Kategorie umschreibt den Leistungs- und Resourcenhunger eines Systems und besteht aus folgenden Unterpunkten [9, S. 7]:

- „Time behaviour“
- „Resource utilisation“
- „Efficiency compliance“

Time Behavior beschreibt die Antwort-, Rechenzeiten und die Durchsatzraten des Systems. Resource Utilisation umfasst die Typen und den Verbrauch von Resourcen. Efficiency Compliance bezieht sich wiederum um die Standard- und Konventionskonformität des Systems. [9, S. 10]

2.3.4 Maintainability

Die Maintainability Kategorie beschreibt, wie einfach ein System gewartet und modifziert werden kann und besteht aus folgenden Unterpunkten [9, S. 7]:

- „Analysability“
- „Changeability“
- „Stability“
- „Testability“
- „Maintainability compliance“

Analysability beschreibt, wie einfach ein System nach Fehlern oder Problemen untersucht werden kann, Changeability wie einfach es ist, Änderungen durchzuführen. Stability besagt, wie wahrscheinlich es ist, dass nach einer Änderung unvorhergesehene Probleme auftreten, Testability wie schwer es ist, das System zu testen. Maintainability compliance, wie auch in den vorherigen Fällen, beschreibt die Standardkonformität. [9, S. 10-11]

2.3.5 Portability

Die Portability Kategorie beschreibt, wie einfach der Wechsel auf andere Hardware- und Softwareumgebungen möglich ist und besteht aus folgenden Unterpunkten [9, S. 7]:

- „Adaptability“
- „Installability“
- „Co-existence“

- „Replaceability“
- „Portability compliance“

Adaptability behandelt die Anpassungsfähigkeit an andere Umgebungen, zB. verschiedene Displaygrößen. Installability befasst sich mit der Einfachheit der Installation. Co-existence beschreibt, wie sich das Programm gegenüber anderen, auf dem gleichen System laufenden Programmen verhält, Replaceability in wieweit das System durch ein anderes ersetzt werden kann. Portability compliance behandelt abschließend die Standardkonformität. [9, S. 11]

3 Softwarearchitektur

Softwarearchitektur beschäftigt sich im Wesentlichen mit dem Erstellen des Fundamentes eines Softwaresystems. Der Fokus liegt nicht auf Implementationsdetails sondern auf Strukturen, logischen und physischen Komponenten, Beziehungen und Schnittstellen. [33, S. 9-10]

Durch diese Abstraktionen wird versucht, die Komplexität des Systems „überschaubar und handhabbar“[33, S. 10] zu machen, um so früh wie möglich Risiken und Probleme zu identifizieren und die Qualität der zu erstellenden Software zu sichern. [33, S. 10]

Diese einzelnen Abstraktionen können in Architektursichten gruppiert werden.

3.1 Architektursichten

Architektursichten beschreiben verschiedene Sichten auf das zu erstellende System und entstanden in den späten 80ern bzw. frühen 90ern. Im Jahre 2000 wurde schließlich eine Empfehlung zur Erstellung von Softwarearchitektur vom Institute of Electronics Engineers (IEEE 1471) heraus gegeben, in welchem Architektursichten erwähnt, aber weder Sichtenmodelle noch Modellierungssprachen festgelegt wurden [6][26, S. 138]. Die Empfehlung wurde im Jahre 2011 von der ISO unter der Nummer 42010 standardisiert[10].

ISO 42010 definiert eine Architektursicht folgendermaßen: Eine Architektursicht ist eine Sicht der Architektur eines Systems aus der Perspektive von ein oder mehreren StakeholderInnen. Diese StakeholderInnen haben unterschiedliche Aufgaben und Interessen und benötigen somit auf ihre Bedürfnisse zugeschnittene Information. Die einzelnen Sichten wiederum beinhaltet mehrere Architekturmodelle. [10]

Ein Beispiel für einen/eine StakeholderIn ist zB. ein/eine SystementwicklerIn. Er/Sie benötigt für das Aufsetzen und Konfigurieren der Hardwarekomponenten eine Sicht auf die physische Verteilung des Systems. Diese Sicht kann zB. mit Hilfe des UML Komponentendiagramms beschrieben werden. Das Komponentendiagramm nimmt in diesem Falle die Rolle des Architekturmodells an.

3.2 Architektursichtenmodelle

ISO 42010 definiert, wie erwähnt, weder Architektursichten- noch Architekturmodelle, stellt jedoch einen Rahmen für diese bereit [26, S. 138]. Diese können, je nach Bedürfnissen der ArchitektInnen, aus einer großen Anzahl von Architektursichtenmodellen ausgewählt werden. [26, S. 142-145]

Folgende Architektursichtenmodelle werden häufig verwendet [33, S. 94]:

- Zachmann Framework
- RM-ODP
- Kruchtens 4+1 Sichtenmodell

Das Zachmann Framework, benannt nach dessen Autor John Zachmann, geht auf den Versuch zurück, Strukturen von Unternehmen zu beschreiben. Das Modell wurde später für Softwarearchitektur abgewandelt. Das Framework eignet sich daher vor allem für Enterprisesysteme, da die Organisation des Unternehmens gut beschrieben werden kann. Für andere Einsatzfälle wird jedoch aufgrund der Komplexität und des Umfangs empfohlen, das Modell auf die exakt benötigten Bereiche einzuschränken. [33, S. 94-95]

RM-ODP, kurz für Reference Model for Open Distributed Processing, ist ein von der ISO standardisiertes Architektursichtenmodell, welches sich vor allem für verteilte, objektbasierte Systeme eignet. Es versucht vor allem Architekturen unabhängig „von Verteilungs- und Implementierungsaspekten“zu machen. [8][33, S. 97-98]

Das 4+1 Schichtenmodell, welches von Philipp Kruchten entwickelt wurde, fokussiert vor allem die ProjektmanagerInnen, EntwicklerInnen und BenutzerInnen. Das Modell beinhaltet folgende Sichten [16][17, S. 501-503]:

- Logical View: Logische Sicht auf das System, welche sich mit der Funktionalität des Systems beschäftigt. Kann durch UML Paket-, Klassen- und Interaktionsübersichtsdiagramme beschrieben werden.
- Process View: Beschäftigt sich mit dem Laufzeitverhalten des Systems, wie zB. Prozessen, Parallelität und Kommunikation. UML Klassen- und Interaktionsübersichtsdiagramme können hierfür verwendet werden.
- Development View: Beschreibt die Aufteilung des Systems in Module und Bibliotheken aus der Sicht des Programmierers. Kann durch UML Paket- und Komponentendiagramme beschrieben werden.
- Physical View: Regelt und beschreibt die Verteilung der Komponenten auf physische Systeme. UML Komponenten- und Verteilungsdiagramm bieten sich hierfür an.
- Scenarios: Beschreibt die Usecases eines Systems, um die Architekturentscheidungen zu validieren. Hierfür können UML Usecasediagramme verwendet werden.

In der Arbeit wird vor allem auf Kruchtens 4+1 Sichtenmodell eingegangen. Aufgrund der Einschränkung auf die Planungsphase der Architektur entfallen jedoch diverse Sichten. Dies sind vor Allem jene, welche sich hauptsächlich mit der Beschreibung der Implementation beschäftigen, zB. die Process View.

3.3 Architekturprozess Lifecycle

Zeitlich ordnet sich die Architekturphase zwischen dem Ermitteln der Anforderungen und der tatsächlichen Implementation ein. Da die Erstellung von Software heutzutage mehr ein agiler und inkrementeller Prozess ist, wird auch die Erstellung und die Mitwirkung der Softwarearchitektur in diese sich wiederholenden Prozesse mit einbezogen. [26, S. 7]

Der Architekturzyklus selbst lässt sich mit dem in Abbildung 2 beschriebenen Aktivitätsdiagramm darstellen und unterteilt sich in folgende Unterbereiche [33, Umschlag]:

- „Erstellen der Systemvision“
- „Verstehen der Anforderungen“
- „Entwerfen der Architektur“
- „Umsetzen der Architektur“
- „Kommunizieren der Architektur“

Die Systemvision beschreibt die Grundidee des zu erstellenden Systems. Sie geht meist vom/von der KundIn aus und wird am Anfang des Projektes erstellt. Eine Systemvision beinhaltet die Antwort auf folgende drei Punkte [30, S. 202-206]:

- Was soll das Projekt erreichen?
- Warum ist es wertvoll?
- Was sind die Kriterien, welche den Erfolg des Projekt beeinflussen?

Bereits hier sollte ein/eine SoftwarearchitektIn anwesend sein, um die Anforderungen zu prüfen, die Machbarkeit zu ermitteln und auf widersprüchliche Anforderungen hinzuweisen. [33, S. 352-353]

Ist die Systemvision erstellt und die Anforderungen ermittelt, müssen diese in eine Form gebracht werden, welche widerspruchsfrei und klar ist. Unpräzise Anforderungen müssen präzisiert werden. Oft findet auch eine Priorisierung der Anforderungen statt. [33, S. 353]

Danach kann mit der Planung der Architektur begonnen werden. Die Anforderungen liefern die Basis für die architektonischen Entscheidungen. In dieser Phase findet meist auch ein Architekturreview statt, um Alternativarchitekturen abzuwägen. [33, S. 353]

Nach der Architekturplanung wird schließlich mit der Implementation begonnen. Hier muss darauf geachtet werden, dass die architektonischen Entscheidungen korrekt umgesetzt werden, da diese erfahrungsgemäß oft vernachlässigt werden. [33, S. 354]

Während der Architekturphase müssen die Entscheidungen des Architekturprozesses kontinuierlich den StakeholderInnen kommuniziert werden, damit diese die Architektur umsetzen und bei auftretenden Problemen Feedback geben können. [33, S. 354]

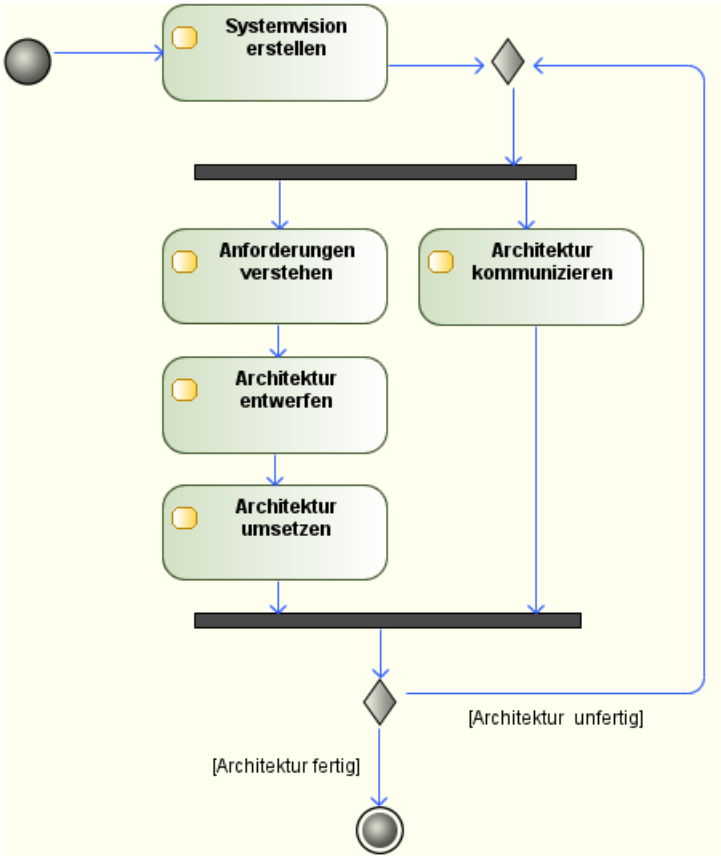

Abbildung 2: Der Architekturprozess modelliert als Aktivitätsdiagramm [33, Umschlag, S. 352]

Der Fokus der Arbeit liegt wegen der fehlenden Implementierung hauptsächlich auf den ersten drei Teilen, der Erstellung der Systemvision, Verstehen der Anforderungen und dem Entwerfen der Architektur. Die Systemvision wurde verwendet, um den Architekturprozess zu erstellen. Bei der Anwendung des Architekturprozesses wird dieser jedoch als Artefakt des Anforderungsprozesses vorausgesetzt.

3.4 Was macht eine gute Softwarearchitektur aus

Die Frage, was eine gute Softwarearchitektur ausmacht, wird oft sehr breit und ungenau beantwortet: Eine gute Architektur erfülle die „Verhaltens-, Qualitäts- und Lebenszyklusanforderun-

gen“ [26, S. 12] eines Systems, ermögliche es Risiken frühzeitig zu erkennen und die inhärente Komplexität eines Systems und dessen Quellcodes zu beherrschen [33, S. 7-8].

Alle diese Kriterien lassen sich jedoch schwierig oder gar nicht messen. Manche Werte sind zwar messbar, aber lassen keine eindeutigen Entscheidungen ableiten: zB. können Kopplung und Kohäsion eines Systems gemessen werden, jedoch kann nicht eindeutig festgelegt werden, ab welchem Wert die Kopplung zu hoch oder die Kohäsion zu niedrig ist.

Dieter Masek kommt zu folgendem Schluss: „Die Frage ob eine gegebene Architektur gut oder schlecht ist, lässt sich nicht direkt beantworten. Gut oder schlecht sind absolute Bewertungen, die für Architekturen sowieso nicht möglich sind. Eine Architektur lässt sich nur in einem festgelegten Kontext beurteilen d.h.: Wie gut löst eine Architektur ein vorgegebenes Problem? Selbst diese eingeschränkte Frage lässt sich nicht mit gut oder schlecht beantworten! “ [20, S. 19]

Trotz all dieser Probleme lassen sich dennoch Werte messen und überprüfen [20, S. 19], welche in Verbindung mit unzureichenden Architekturentscheidungen gebracht werden können. Dies erlaubt folgenden Schluss: Die Ausgangsfrage, was eine gute Softwarearchitektur ausmacht, führt zu keiner Erkenntnis und ist damit mehr oder minder nutzlos: Es kommt nicht darauf an, ob eine Architektur gut ist, sondern ob eine Architektur die an sie gestellten Anforderungen erfüllt.

Viele dieser Anforderungen stehen im Gegensatz zueinander: zB. erfordert eine hohe Performance eine niedrigere Abstraktionsebene, was jedoch die Wartung des Codes erschwert. Die Erstellung einer angemessene Softwarearchitektur beschäftigt sich somit mehr mit der Abwägung von Vor- und Nachteilen, welche aus den jeweiligen Entwurfsmustern und Technologien ableitbar sind.

Um diese Entscheidungen abwägen zu können, ist wichtig, die expliziten Anforderungen an das System zu kennen [20, S. 19]. Eine möglichst vollständige Anforderungsanalyse ist somit die Ausgangsbasis für eine angemessene Softwarearchitektur. Dies kann jedoch vor Allem am Anfang der Anforderungsanalyse schwierig sein, da die KundInnen noch keine genaue Vorstellung vom zu entwickelnden System haben [28, S. 80].

Eine Möglichkeit zur Überprüfung von bestehenden und noch nicht beachteten Anforderungen der Architektur stellt die Durchführung eines szenariobasierten Architekturreviews wie ATAM oder CBAM dar.

3.5 Architekturbewertungsmethoden

Architekturbewertungsmethoden werden dazu verwendet, die Angemessenheit einer Architektur zu überprüfen. Die Meisten ihrer Art versuchen dies mit Hilfe von Szenarien zu lösen, vor allem weil das Kosten-Nutzen Verhältnis dieser Herangehensweise sehr gut ist [26, S. 185] und Szenarien oft schon im Anforderungsprozess erstellt wurden[32, S. 62].

Es gibt mehrere verschiedene Bewertungsmethoden wie CBAM, ALMA, ARID, uvm. welche zum Großteil Abwandlungen von ATAM darstellen. ATAM wiederum ist eine Weiterentwicklung

von SAAM. [20, S. 60-76]

Da Kosteneffektivität, Änderungs- und Wachstumsszenarien die Basis für Entscheidungen des erstellten Architekturprozesses darstellen, wird vor Allem näher auf ATAM (Änderungs- und Wachstumsszenarien) und CBAM (Kostenszenarien) eingegangen.

3.5.1 ATAM

ATAM, kurz für Architecture Trade-off Analysis Method, beschäftigt sich mit den nicht funktionalen Anforderungen wie Performance oder Verfügbarkeit. Sie lässt sich in folgende Phasen einteilen [26, S. 185]:

- Vorbereitungsphase
- Architekturzentrierte Bewertung
- StakeholderInnenzentrierte Bewertung
- Nachbearbeitungsphase

Einer initialen Vorbereitungsphase, in welchem das Team und der Zeitplan erstellt wird, folgt ein Treffen mit den StakeholderInnen des/der AuftraggeberIn, in der die Geschäftsziele und Architektur vorgestellt werden. In diesem Treffen werden die nicht funktionalen Anforderungen anhand eines Utility Trees (Abbildung 3) erarbeitet, priorisiert und bewertet. Sind wichtige architekturrelevante Anforderungen noch nicht bekannt, können diese nun mit den StakeholderInnen erarbeit werden. [26, S. 184-199]

Der Utility Tree ist ein Baumstruktur, deren Blätter die nicht funktionalen Anforderungen sind. Diese nicht funktionalen Anforderungen werden danach in weitere, konkrete Unterbereiche aufgespalten, zB. Performance würde in Antwortzeiten und Speicherbedarf aufgespalten werden. Diese Unterbereiche können schließlich bewertet und priorisiert werden. Die Bewertung und Priorisierung erfolgt anhand der Wichtigkeit und der Risiken.

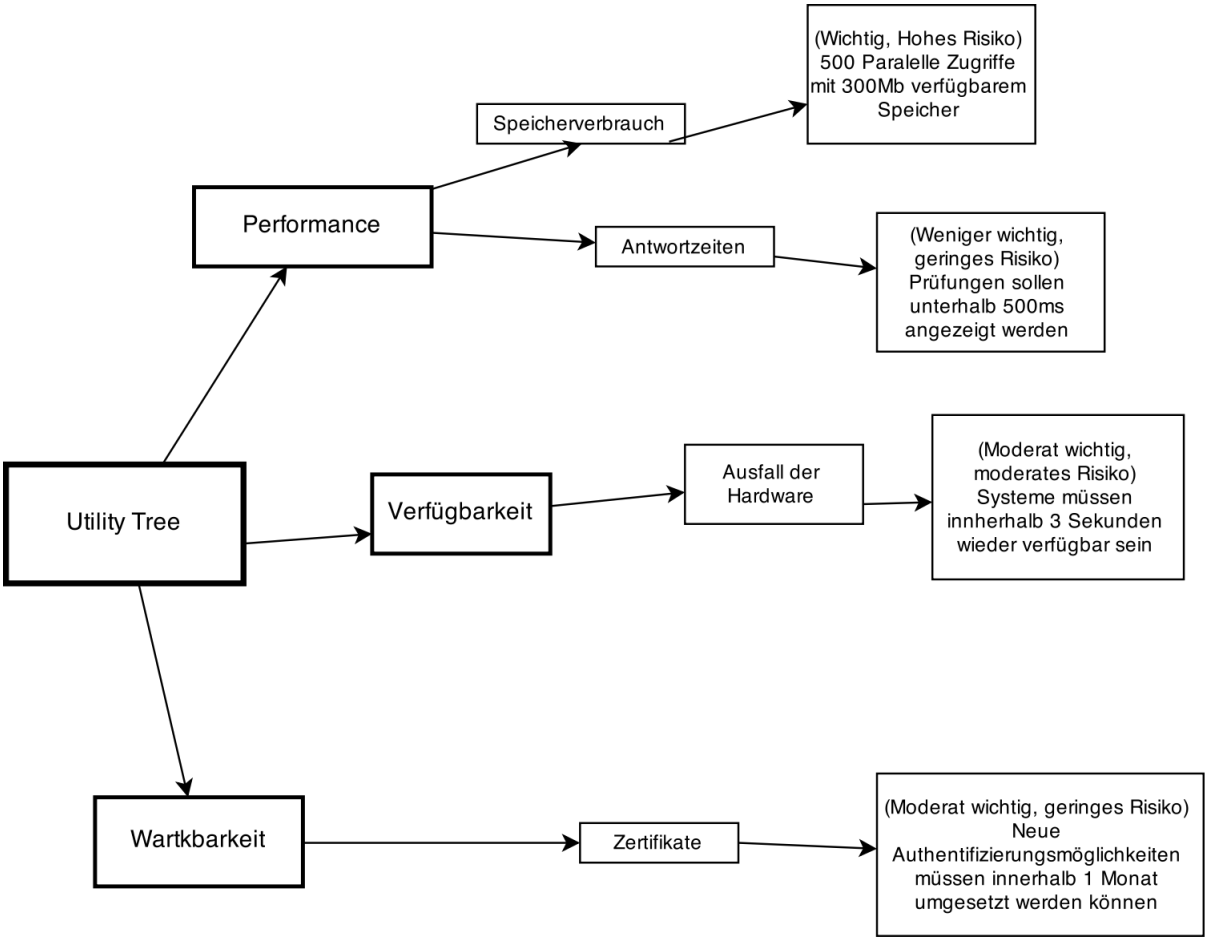

Abbildung 3: Beispiel eines Utility Trees [15, S. 17]

Auf Basis des Utility Trees und dessen nicht funktionale Anforderungen werden dann in den darauf folgenden Bewertungsphasen folgende Szenarientypen überprüft [20, S. 62-67][26, S. 188]:

- Usecases
- Wachstumsszenarien
- Explorative Szenarien

Die Usecases beschreiben die Standardanwendungsfälle der NutzerInnen, welche durch das System umgesetzt werden sollen. Wachstumsszenarien beschreiben Szenarien, welche sich vor allem mit der Skalierbarkeit der Applikation beschäftigen, zB. statt 10 UserInnen greifen auf einmal 100 UserInnen auf das System zu. Explorative Szenarien drehen sich um mögliche Nutzungsszenarien, welche im Vorfeld entweder nicht bedacht wurden, oder nicht im Hauptfokus des Systems stehen. [20, S. 63]

Alle Szenarien werden priorisiert. Danach führen die ArchtektInnen pro Usecase eine Analyse der im Utilitytree beschriebenen nicht funktionalen Anforderungen durch. Je nachdem, wie

viel Zeit für die Analyse aufgewendet werden soll, werden zusätzlich zu den als hoch priorisierten Usecases und nicht funktionalen Anforderungen auch weitere, niedriger Priorisierte überprüft. [26, S. 192]

Sind die Bewertungsphasen abgeschlossen können die Ergebnisse analysiert, dokumentiert und präsentiert werden. Dies dient nicht nur als Dokumentation der geplanten Verbesserungen für die Architekturplanung, sondern hilft auch in zukünftigen ATAM Analysen die Entscheidungen früherer Analysen nachzuvollziehen. [26, S. 189]

3.5.2 CBAM

Eine Architektur ist meist nicht nur durch die ihr abverlangten Qualitätsanforderungen begrenzt, sondern auch durch die ihr zur Verfügung stehenden Mittel. Es mag zB. besser sein, eine hohe Kohäsion einer Komponente zu verlangen, um die Wartungskosten zu verringern, jedoch kann dies in manchen Fällen zu zu hohen Anschaffungskosten führen, da zB. statt einem Server zehn Server angeschafft werden müssten. Um die Kosten verschiedener Architekturen besser bewerten zu können, wurde CBAM entwickelt . CBAM, kurz für Cost Benefit Analysis Method, baut auf ATAM auf und ergänzt es um eine Kosten-Nutzen Analyse. Hauptkriterium dieser Analyse ist der ROI. [20, S. 67-68]

CBAM gliedert sich in folgende Phasen: [20, S. 68]

- „Triagephase“
- „Detailphase“
- „Attributsquantitfizierungsphase“
- „Kostenquantitfizierungsphase“
- „Rankingphase“

In der Triagephase werden die möglichen Architekturen anhand ihrer Kosten und Vorteile grob bewertet. Die Bewertungen liefern dann die Grundlage für die Auswahl der möglichen Architektur(en). Bevorzugt wird die Architektur, welche die höchste Wertschöpfung erlaubt, sprich welche den ROI maximiert. [20, S. 68]

In der darauf folgenden Detailphase wird dann die ausgewählte Architektur und deren Qualitätsanforderungen genauer analysiert, um in der Attributsquantitfizierungsphase eine Aussage über die Auswirkungen auf die Qualitätsanforderungen treffen zu können. In der Kostenquantitfizierungsphase wird dann der ROI der verbleibenden Architekturen berechnet. [20, S. 68-69]

Abschließend wird nun auf Basis des ROI versucht, die Architektur mit dem größten ROI zu finden, welche das Budget für das System erlaubt. Im Gegensatz zu ATAM wird kein Utility Tree verwendet, sondern eine „Utility-Response-Kurve“, welche Auskunft über die Wirtschaftlichkeit der Qualitätsanforderungen liefert. [20, S. 69]

4 Modellierung in der Architektur

Der Architekturprozess „erstreckt sich von der Analyse des Problembereichs eines Systems bis hin zu seiner Realisierung“[33, S. 10] und arbeitet durch die verschiedenen Architektursichten stark mit verschiedenen Mitteln der Abstraktion. Wien in ISO 42010 beschrieben besitzt jede Architektursicht zudem ein oder mehrere Architekturmodelle [10]. Werden Architektursichten im Architekturprozess verwendet, bietet sich deswegen auch die Verwendung von Modellierungssprachen an, mit welchen die Architekturmodelle erstellt werden können.

4.1 Modelldefinition

Ein Modell hat laut Stachowiak folgende Eigenschaften [31, S. 131-133]:

- Abbildung
- Verkürzung
- Pragmatismus

Das Abbildungsmerkmal besagt, dass Modelle entweder natürliche oder künstliche Dinge abbilden. Das Verkürzungsmerkmal eines Modells dreht sich um die Reduktion der Eigenschaften des Originals: Nur die für den Einsatzbereich relevanten Eigenschaften werden durch das Modell repräsentiert. Das Pragmatismusmerkmal schließlich besagt, dass Modelle eine Funktion erfüllen: Sie sind sowohl für einen definierten Einsatzzweck, als auch für eine Person, sei sie natürlich oder künstlich, erstellt. [31, S. 131-133]

Werden Modelle verwendet, müssen diese kommuniziert und dokumentiert werden [19, S. 12]. Vor Allem in der Arbeit im Team bietet sich deswegen eine standardisierte Modellierungssprache an, welche den Einsatzbereich so gut wie möglich abdeckt. Dies dient vor Allem dazu, Interpretationsfehler zu vermeiden. Aus dem gleichen Grund ist es auch vorteilhaft, eine weit verbreitete Modellierungssprache zu wählen [32, S. 139]. Eine Modellierungssprache sollte zudem visuelle Elemente verwenden, da das menschliche Auge Bilder schneller als Text aufnehmen kann [19, S. 12].

4.2 UML

UML, kurz für Unified Modelling Language, ist eine in den 90ern entstandene Modellierungssprache, welche sich vor Allem für die Modellierung objektorientierter Systeme eignet [26, S. 145]. Sie war eine Antwort auf den damaligen Wildwuchs an verschiedenen, zueinander inkompatiblen Modellierungssprachen wie OOD, OOA&D, etc. [29, S. 5]. Mittlerweile ist sie weit verbreitet und wird weltweit verstanden [32, S. 138]. Die aktuelle Version der UML ist 2.4.1 [24].

Die in der UML spezifizierten Diagramme lassen sich in folgende zwei Kategorien einteilen [29, S. 105, 239][26, S. 146]:

- Verhaltensdiagramme
- Strukturdiagramme

UML Diagramme lassen sich im Architekturprozess vor Allem zur Beschreibung der verschiedenen Architektursichten einsetzen und bieten alle für die Abstraktion des Systems im Architekturprozesses benötigten Werkzeuge[32, S. 139]. Durch ihre verkürzende Wirkung und höhere Informationsdichte sind UML Diagramme auch oft Artefakte des Anforderungsprozesses [28, S. 215]. UML schlägt somit die Brücke vom Anforderungs- zum Architekturprozess. Nicht alle verfügbaren UML Diagramme sind jedoch für die Erstellung und Modellierung der Architektur nötig. [26, S. 144]

Die in der Arbeit verwendeten Verhaltensdiagramme beschränken sich auf das Usecase-, Kontext-, Komponenten- und Aktivitätsdiagramm. Das einzig verwendete Strukturdiagramm ist das Klassendiagramm.

4.2.1 Usecasediagramm

Das Usecasediagramm gliedert sich in die Diagrammkategorie der Verhaltensmodellierung ein und beschreibt die Anwendungsfälle des Systems, dessen AkteurInnen und die Beziehung zwischen den Beiden. Die einzelnen Anwendungsfälle selbst repräsentieren Aktionen. Diese werden jedoch nicht im Usecasediagramm modelliert.[29, S. 242-245]

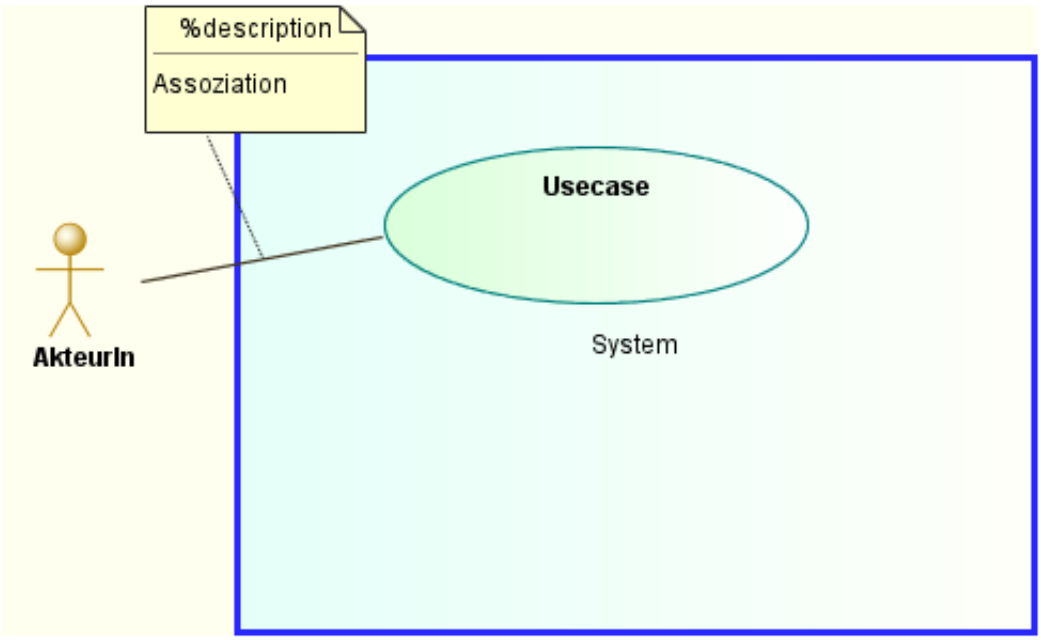

Abbildung 4: Das UML Usecasediagramm

Das Usecasediagramm ist kein Artefakt der Architekturerstellung, sondern wird in der Regel in der Anforderungsanalyse des Projektes erstellt. In der Architekturphase gibt das Diagramm aber Auskunft über wichtige architekturentscheidende Parameter, wie zB. die Systemabgrenzung, Nachbarsysteme, BenutzerInnen und Grundfunktionalität des Systems. Außerdem wird es für Architekturreviewmethoden wie ATAM benötigt und kommt in der Usecase Sicht von Kruchtens 4+1 Modell vor [26, S. 148]

4.2.2 Kontextdiagramm

Das Kontextdiagramm kann in die Strukturmodellierung eingeordnet werden und zeigt die AkteurInnen, Datenflüsse und Nachbarsysteme. Das Kontextdiagramm selbst ist nicht Teil der UML, kann aber mit UML dargestellt werden. Meistens wird dafür ein Usecasediagramm verwendet, aber auch ein Komponentendiagramm stellt im Prinzip die notwendigen Elemente bereit. [29, S. 255]

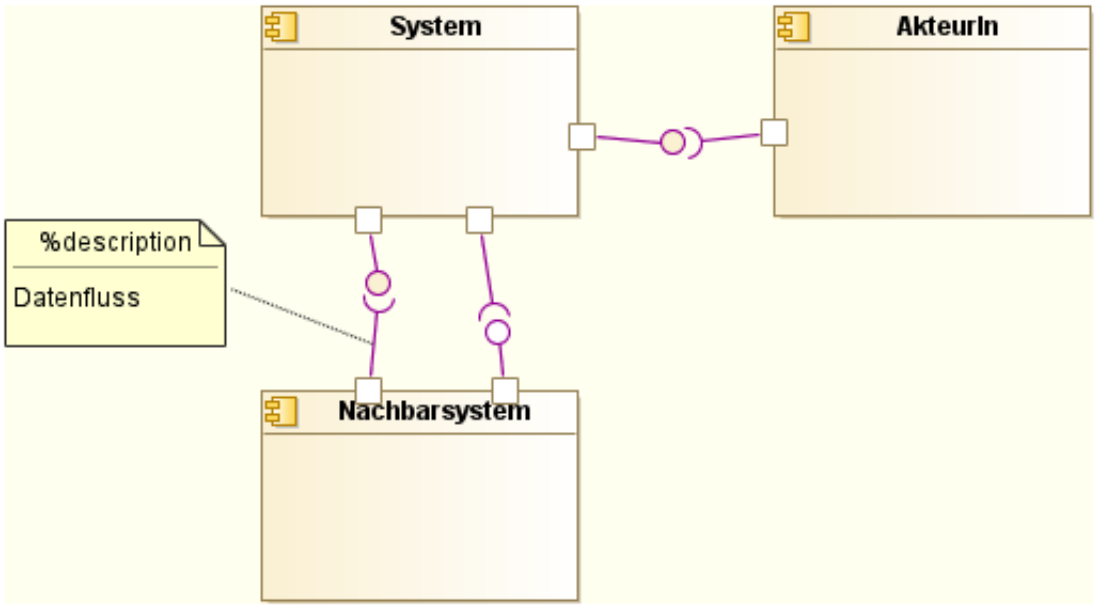

Abbildung 5: Das Kontextdiagramm, dargestellt mit Hilfe des Komponentendiagramms

Das Kontextdiagramm wird in der Anforderungsanalyse erstellt und gibt einen genaueren Überblick über die Systemabgrenzung [29, S. 255]. Da es zudem auch die Nachbarsysteme modelliert kann es als Grundstein für die Planung der Komponenten der Architektur herangezogen werden.

4.2.3 Komponentendiagram

Das Komponentendiagramm ist Teil der Strukturdiagramme und wird verwendet, um die Bestandteile eines Systems zu modellieren. Die Implementation der einzelnen Bestandteile, auch

Komponenten genannt, wird dabei nicht dargestellt. Stattdessen werden die Schnittstellen der Komponenten und deren Beziehungen untereinander modelliert. Die Komponenten selbst können in Artefakte gekapselt werden. [29, S. 216]

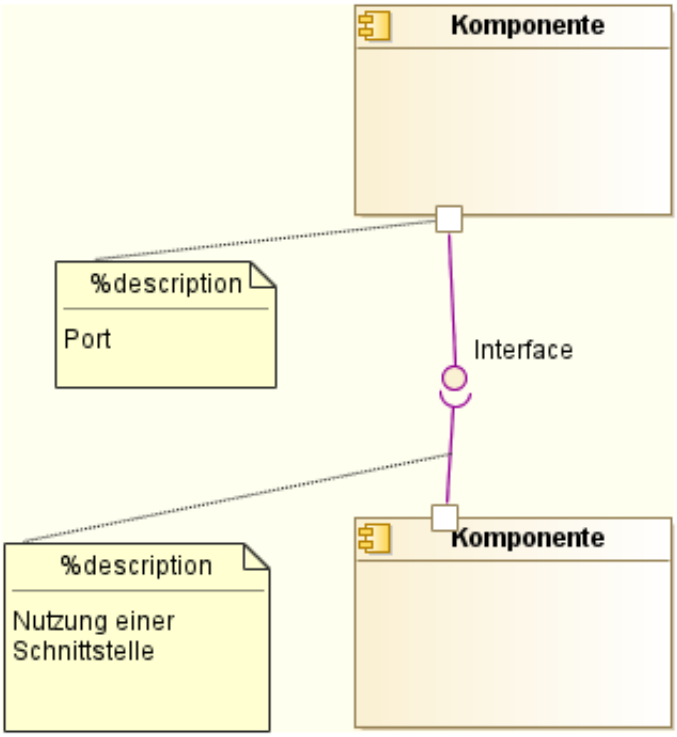

Abbildung 6: Das UML Komponentendiagramm

Das Komponentendiagramm modelliert im Gegensatz zum Paketdiagramm bis zu einem gewissen Grad die tatsächliche, physische Aufteilung des Systems und kann in der Architekturplanung deswegen für die Physische bzw. Verteilungssicht verwendet werden [29, S. 223][26, S. 139]. Durch die Verwendung von Interfaces ermöglicht es eine grobe und frühe Übersicht über das Zusammenspiel des zu entwickelnden Systems.

4.2.4 Klassendiagramm

Das Klassendiagramm fügt sich in die Strukturdiagramme ein und modelliert die Daten, Methoden und deren Beziehungen untereinander. Es kann als Basis für die Erstellung der Datenbanktabellen verwendet werden und übernimmt somit immer öfter die Aufgaben, welche früher dem ER Diagramm zukamen. Eine weitere Aufgabe des Klassendiagramms ist die Modellierung von Interfaces. [29, S. 108-110]

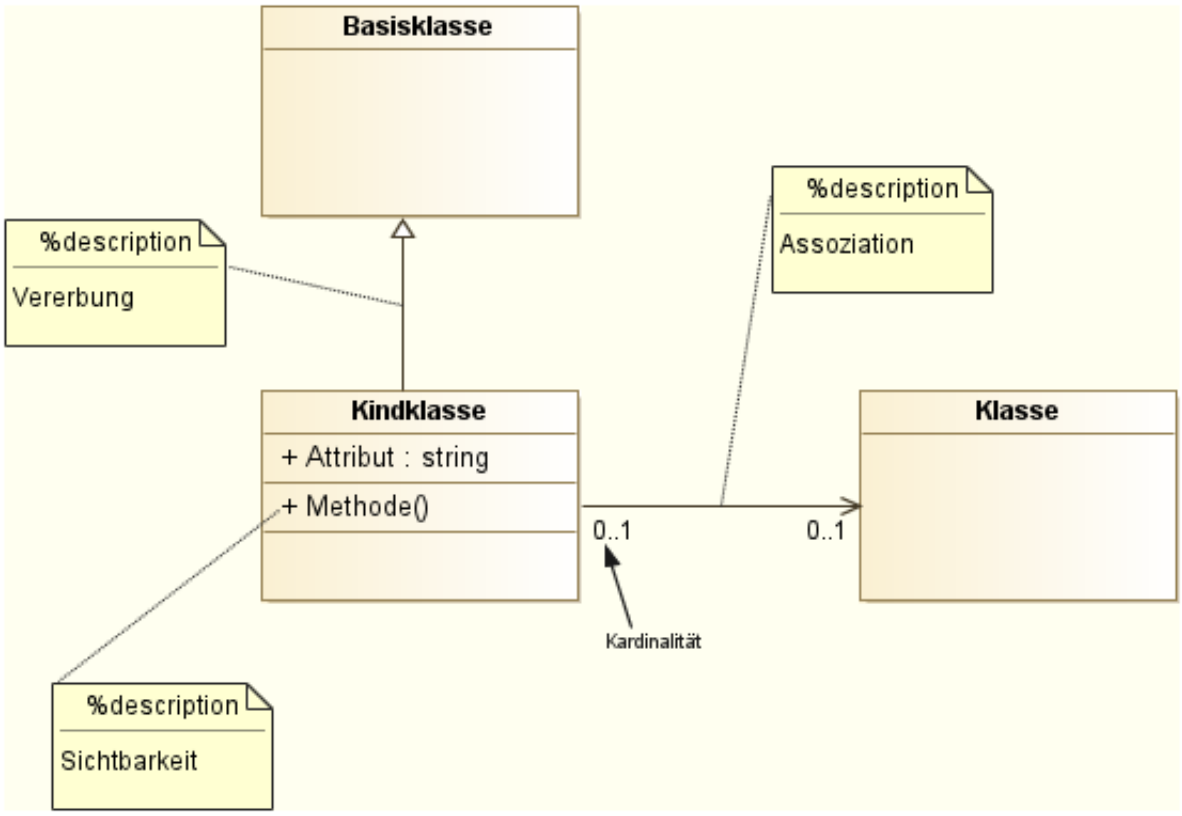

Abbildung 7: Das UML Klassendiagramm

Da die Daten und Interfaces, welche das Klassendiagramm beschreibt, auch in anderen Diagrammen referenziert werden, sind sie ein wichtiger Teil der Architekturerstellung: Interfacediagramme werden für die Beschreibung der Schnittstellen von Komponenten benötigt, Klassendiagramme werden unter Anderem in den Objektflüssen der Aktivitätsdiagramme referenziert. Außerdem lassen sich die Daten für den/die KundIn mit einem Wert beziffern, da die Vertrautheit der Daten durch Gesetze vorgeschrieben sein kann und auch ihr Verlust, Manipulation oder Diebstahl meist finanzielle Folgen nach sich zieht.

4.2.5 Aktivitätsdiagramm

Das Aktivitätsdiagramm wird in die Verhaltensmodellierung eingeteilt und erlaubt es, Abläufe des Systems zu modellieren. Es besteht aus Aktionen, Objektknoten, Kontrollelementen und Kanten, welche die Elemente untereinander verbinden [29, S. 264]. Durch Objektflüsse und Swimlanes lässt sich der Austausch von Daten zwischen verschiedenen Systemen modellieren. [29, S. 268]. Diese Flüsse können auch als Nachrichten im Sequenzdiagramm modelliert werden. Das Sequenzdiagramm ist aber zeitraubender zu modellieren, was dazu führt, dass die Wartung oft vernachlässigt wird [29, S. 414]. [29, S. 263-274]

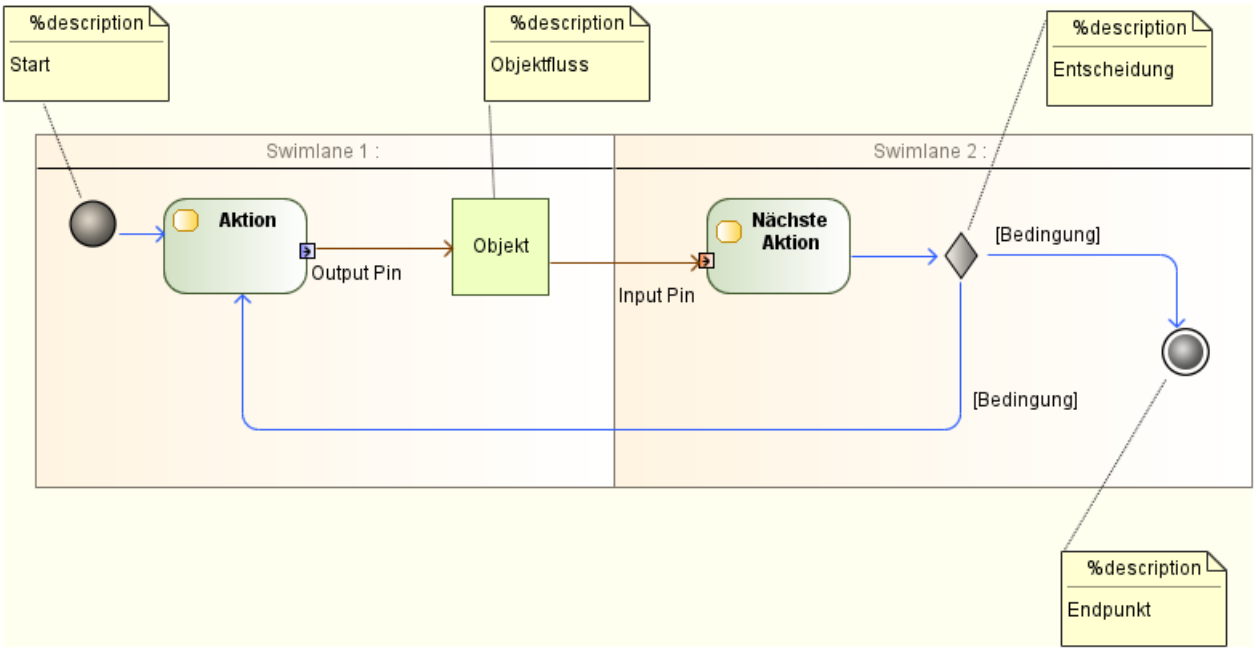

Abbildung 8: Das UML Aktivitätsdiagramm

Da sich das Aktivitätsdiagramm vor Allem zur Beschreibung von Businessprozessen und Usecases eignet, ist es bei der Architekturerstellung vor Allem für das Verständnis des Usecasediagrammes wichtig [29, S. 271-272]. Zusätzlich helfen die modellierten Objektflüsse und Swimlanes bei der Erstellung der Komponenteninterfaces.

5 Prozesserstellungsversuche

Der Architekturprozess ist komplex und ein falsches Vorgehen kann der Ursprung vieler Probleme sein [33, S. 7-8]. Deswegen ist es notwendig einen eigenen Prozess zu definieren, welcher die Erstellung vereinfacht und die Fehlerkosten minimiert. Dieser Prozess sollte schon in der Planungsphase zum Einsatz kommen, da hier wegen der Zehner-Regel der Fehlerkosten der größte Effekt zur Reduzierung der Fehlerkosten erzielt werden kann [25, S. 154].

Der Prozess wurde anhand eines Beispielprojektes erstellt und dreht sich um die Architektur eines Systems einer Personenzertifizierungsstelle.

5.1 Vorhandene Daten

Ausgegangen wurde von folgenden Anforderungsdokumenten, welche in der Erstellung des Prozesses mehrfach abgeändert und an die Architekturprozessanforderungen angepasst wurden:

- Usecasediagramm: modelliert die Usecases des Unternehmens
- Usecasebeschreibung: ausgefülltes Anforderungstemplate, welches Sonderfälle, nicht funktionale Parameter und weitere Details beinhaltet
- Klassendiagramm: visualisiert die zu verwendeten Daten
- Aktivitätsdiagramme: visualisiert den Ablauf komplexerer Usecases
- Kontextdiagramm: zeigt die Datenflüsse zwischen AkteurInnen, Nachbarsystemen und dem zu erstellenden System
- ISO Anforderungsdokument für Personenzertifizierungsstellen [12]: beschreibt die Rahmenbedingungen für den Betrieb einer Personenzertifizierungsstelle

5.2 Prozesserstellungsversuche

Ausgehend von den vorhandenen Daten wurden mehrere Prozesse definiert, welche bis auf den Letzten entweder zu grobe Ergebnisse lieferten, oder nicht nachvollziehbar waren.

Ausgangsbasis war eine Systemvision mit folgenden Anforderungen:

- Es soll eine Webseite entstehen, welche die Prüfungstermine auflistet und Personen erlaubt, sich für diese Prüfungen anzumelden. Dadurch soll der Verwaltungsaufwand reduziert werden, um Kosten zu sparen.
- Die Übermittlung der Prüfungsdaten soll über einen eigenen VPN Server geschehen, um die Datensicherheit des Systems zu erhöhen.
- Die Prüfungsdaten werden firmenintern verwaltet und nach der Auswertung soll der Scheme Owner benachrichtigt werden. Beide Usecases sollen auf eine sichere Art und Weise umgesetzt werden.

Aufbauend darauf wurde dann versucht einen Prozess zu finden, der diese Grundideen berücksichtigt.

Die anfänglichen Versuche basierten stark auf einem ATAM Utility Tree ähnlichen Verfahren, bei welchem die nicht funktionalen Anforderungen nach der Formel von Oliver Vogel priorisiert wurden: „Priorität = (Nutzen + Risiko + Wirkung) / 3“[33, S. 374]. Die Bewertung der Komponenten wurde auf Basis einer bestehenden Tabelle mit Basisarchitekturen abgeleitet [20, S. 179].

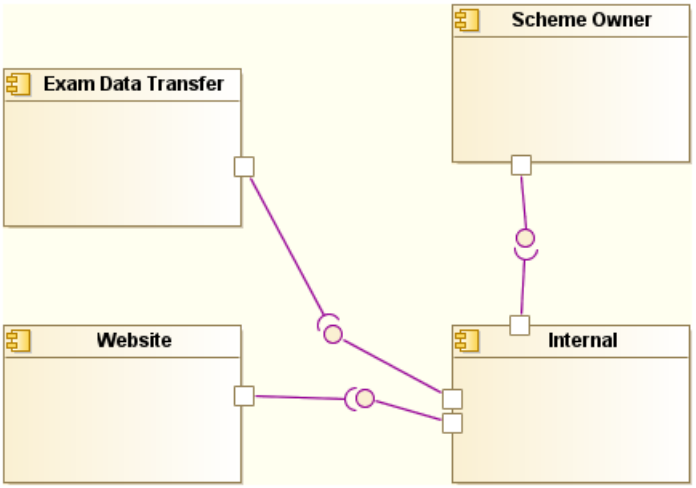

Abbildung 9: Systemvision der Komponenten

5.2.1 Vom Usecase zur Komponente durch Priorisierung der nicht funktionalen Anforderungen

Der erste Versuch zur Erstellung des Architekturprozesses orientierte sich am Prinzip: teile und herrsche. Der Prozess verwendete aufgrund der initialen Vermutung, dass nicht funktionalen Anforderungen die Hauptentscheidungsbasis für die Architektur darstellen, eine priorisierte Liste von nicht funktionalen Anforderungen. Danach wurde versucht, anhand der priorisierten Anforderungen passende Komponenten auszuwählen. Der genaue Ablauf war wie folgt:

- Für jeden Usecase wird ein komplettes Komponentendiagramm des Systems erstellt.
- Die Komponenten jedes Teilsystems werden anhand ihrer nicht funktionalen Qualitäten aus einem Pool von Komponentenarchitekturen gewählt. Diese Komponentenarchitekturen beinhalteten zB. Systeme wie den üblichen Webstack, welcher sich aus Komponenten wie dem Loadbalancer, Datenbankserver, Applikationsserver und Webserver zusammensetzt.
- Schlussendlich werden alle Teilsysteme miteinander vereinigt, soweit es die nicht funktionalen Attribute erlauben.

Dieser Prozess scheiterte nicht nur am enormen Modellierungsaufwand, sondern auch am Auswahlprozess der Komponentenarchitekturen: Je nachdem, welche Komponentenarchitekturen vorhanden waren und wie diese bewertet wurden, entstanden unterschiedliche Architekturen. Zudem schien es zu viele Komponentenarchitekturen zu geben, da einzelnen Komponenten beliebig miteinander kombinierbar waren.

Die Qualität der Architektur hätte folglich von der Vollständigkeit dieser scheinbar unendlich großen Menge an Komponentenarchitekturen abgehangen. Aus diesem Grund schien der Prozess ungeeignet für die Architekturerstellung und wurde somit verworfen.

5.2.2 Von einer Architektur mit hoher Kohäsion und anschließendem Architekturreview zu den Komponenten

Um das Problem des sehr hohen Modellierungsaufwandes des ersten Prozesses zu umgehen, wurde von einer Grundarchitektur mit hoher Kohäsion ausgegangen. Diese Komponentenarchitektur entstand zusammen mit dem/der AuftraggeberIn, um zusätzliche Risiken indentifizieren zu können.

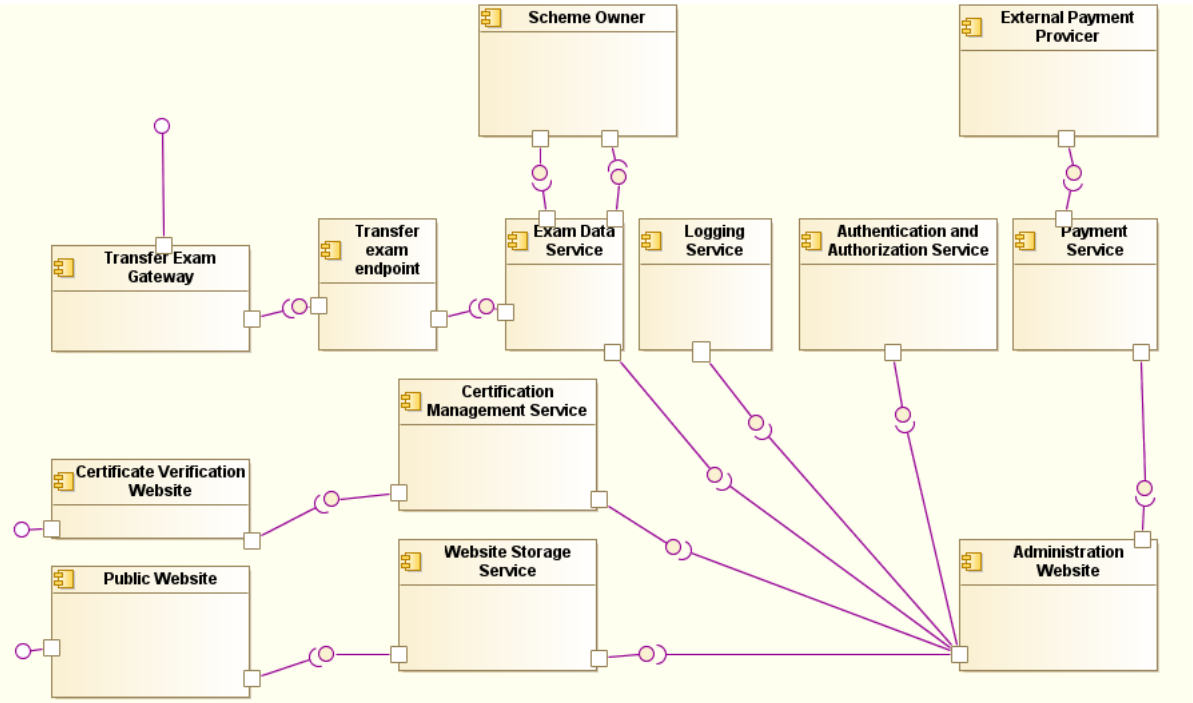

Abbildung 10: Architektur mit hoher Kohäsion

Diese Architektur sollte nun auf die Erfüllung der nicht funktionalen Anforderungen überprüft und gegebenenfalls angepasst werden. Zur Überprüfung der Anforderungen wurden die priorisierten, nicht funktionalen Anforderungen eines Usecases herangezogen. Der Prozess ähnelte damit stark einem szenariobasierten Review wie er zB. in ATAM durchgeführt wird.

Auch dieser Prozess litt jedoch unter dem Problem, dass die nicht funktionalen Anforderungen schwer bewertet werden konnten. Außerdem war es schwer ein Regelwerk/Rezept aus der Architekturerstellung abzuleiten, da durch die Einbeziehung unterschiedlicher AuftraggeberInnen jeweils verschiedene Architekturen entstehen können. Die Einbeziehung von Kohäsion als Aufteilungsgrundlage der Komponenten verursachte zudem ein gefühlt zu großes System, welches in der Umsetzung sehr teuer geworden wäre. Die Kostenersparnis des Systems, welche in der Systemvision definiert wurde, schien damit unzureichend erfüllt zu werden.

5.2.3 Von den Daten zu den Komponenten

Die Auswahl und Bewertung der Komponentenarchitekturen und der starke Fokus auf die nicht funktionalen Anforderungen in den beiden vorherigen Prozessen stellte ein wesentliches Hindernis zur Erstellung eines eindeutigen Regelwerkes dar: Eine vollständige Auflistung aller möglichen Komponentenarchitekturen erschien entweder unmöglich oder unvollständig zu sein; eine Bewertung der nicht funktionalen Attribute schien ohne entsprechende Implementation nur sehr grob überprüfbar zu sein. Der Versuch, ein System mit hoher Kohäsion zu erstellen, endete zudem in sehr teuren Architekturen.

Dies war überraschend, da die populärste Architekturbewertungsmethode, ATAM, stark auf nicht funktionale Anforderungen aufbaute. Als Grund für diese Inkompatibilität wurde der Zeitpunkt der Architekturerstellung vermutet: Durch die fehlende Implementationsphase waren die nicht funktionalen Anforderungen sehr schwer zu bewerten und somit mehr oder weniger nicht überprüfbar. Deshalb wurden sie als Hauptkriterium und Ausgangspunkt für die Architekturerstellung verworfen.

Stattdessen wurde der Fokus auf die Aufspaltung der Daten gelegt. Die Daten wurden anhand Ihrer Vertraulichkeit in unterschiedliche Netzwerke aufgeteilt. Diese Netzwerke wurden dann durch Komponenten miteinander verbunden, die den Zugriff auf die Daten regelten.

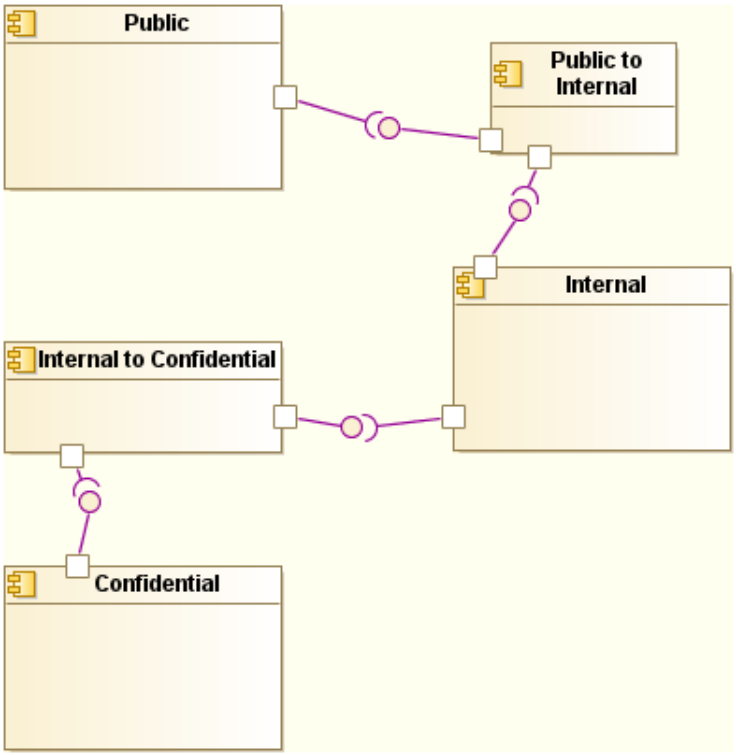

Abbildung 11: Aufteilung der Komponenten in Datenbereiche

Dieser Prozess erlaubte es, eine nachvollziehbare Architektur zu erstellen, jedoch war das Ergebnis zu grob. Außerdem schien eine separate Komponente zur Übertragung der Prüfungs-

daten zu fehlen, welche in der ursprünglichen Systemvision definiert und als notwendig empfunden worden war, um die Rahmenbedingungen der Vertrautheit zu erfüllen [12, 7.3].

5.2.4 Von den Daten und den AkteurInnen zu den Komponenten

Aufbauend auf dem vorherigen Prozess, welcher die Architektur anhand der Daten erstellte, wurden nun auch AkteurInnen eingebunden und deren Beziehungen zu den Daten ermittelt. Anhand dieser Beziehungen wurden Regeln erstellt, aus denen wiederum die Architektur erstellt wurde.

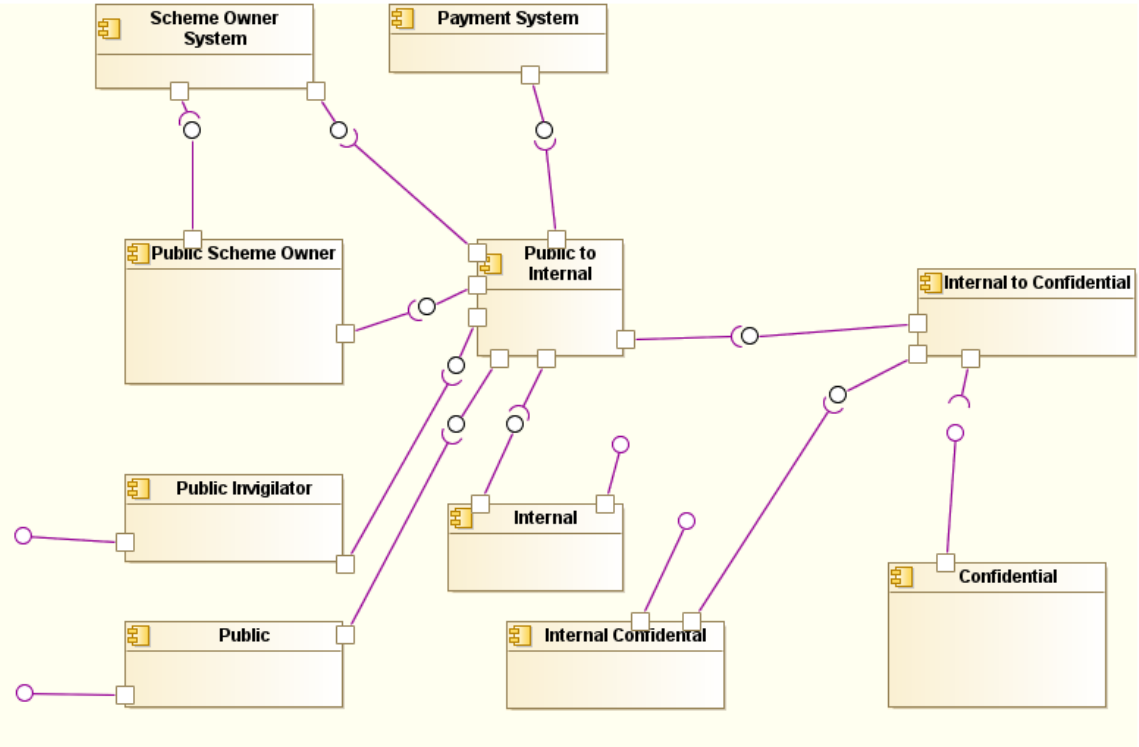

Abbildung 12: Aufteilung der Komponenten in Datenbereiche und AkteurInnen

Dieser Prozess schien nicht nur die Rahmenbedingungen und Sicherheitsbedingungen abzudecken, er war auch durch die erstellten Regeln nachvollziehbar und genau genug, um bereits einen guten Blick auf die Architektur zu erlangen. Anhand der daraus resultierenden Architektur war es nun auch möglich, nicht funktionale Attribute wie zB. Antwortzeiten besser einschätzen zu können.

6 Ermittlung der Architekturanforderungen

Da die Software Architektur auf den Anforderungen basiert und viel wichtiger „den gestalterischen Spielraum des Architekten“[33, S. 103] begrenzt, kann davon abgeleitet werden, dass sowohl die Qualität der Architektur als auch die Akzeptanz des Systems wesentlich von den bereits im Vorfeld ermittelten Parametern abhängt. Das bedeutet wiederum, dass für die Klärung der Ausgangsfrage - Wie kommt man von Anforderungen auf eine gute Architektur - auch der Anforderungsprozess eine wichtige Rolle spielt.

Da der Anforderungsprozess ein an sich eigenes, sehr großes Themengebiet darstellt, wird hier jedoch nur auf die Ausgangsartefakte eingegangen, welche später im Architekturprozess referenziert werden. Zu diesen Parametern zählen nicht nur die Daten, AkteurInnen und Usecases, sondern auch die vorgegebenen nicht funktionalen Anforderungen.

Die Entwicklung der meisten Software Projekte ist oft ein dynamischer und agiler Prozess. Während des Entwicklungsprozesses werden oft Änderungen eingebracht und neue Anforderungen aufgestellt [32, S. 6-7, S. 37]. Auch führt die Komplexität des Systems und das Wissen des/der KundIn dazu, dass nach der Erhebung der initialen Anforderungen nicht alle Parameter vollständig bekannt sind. Dies wiederum führt mehr oder weniger dazu, dass der Anforderungsprozess während der Entwicklung der Architektur nicht als abgeschlossen gesehen werden kann und der/die KundIn verfügbar sein muss.

Aufgrund der Zehner-Regel der Fehler [25, S. 154] ist es zwar wichtig, möglichst viele Parameter schon so früh wie möglich zu kennen, jedoch führt dies in manchen Fällen zu einem verhältnismäßig zu hohem Aufwand. Zum Beispiel ist es wichtig und interessant die Schadenskosten zu kennen, welche nach einem unberechtigten Zugriff oder Manipulation von Daten durch die AkteurInnen auftreten können, jedoch erfordert dies eine komplette Gegenüberstellung von allen Daten und AkteurInnen in sämtlichen möglichen Szenarien: Bearbeiten, Erstellen, Löschen und Lesen. Ein Großteil dieser Kombinationen kann jedoch schon nach einer initialen, kurzen Phase der Aufspaltung in Netzwerke ausgeschlossen werden, weil bestimmte AkteurInnen keinen Zugriff mehr auf bestimmte Komponenten haben. Deswegen wird auf diese Szenarien erst nach einer kurzen, initialen Architekturphase eingegangen.

6.1 Ermittlung der Usecases

Die Usecases werden zusammen mit dem/der KundIn ermittelt. Daraus wird schlussendlich ein Usecasediagramm erstellt, welches alle AkteurInnen und Nachbarsysteme beinhaltet. Dies ist wichtig für das Kontextdiagramm, welches auch im Anforderungsprozess erstellt wird und die Ausgangsbasis für die Architektur darstellt.

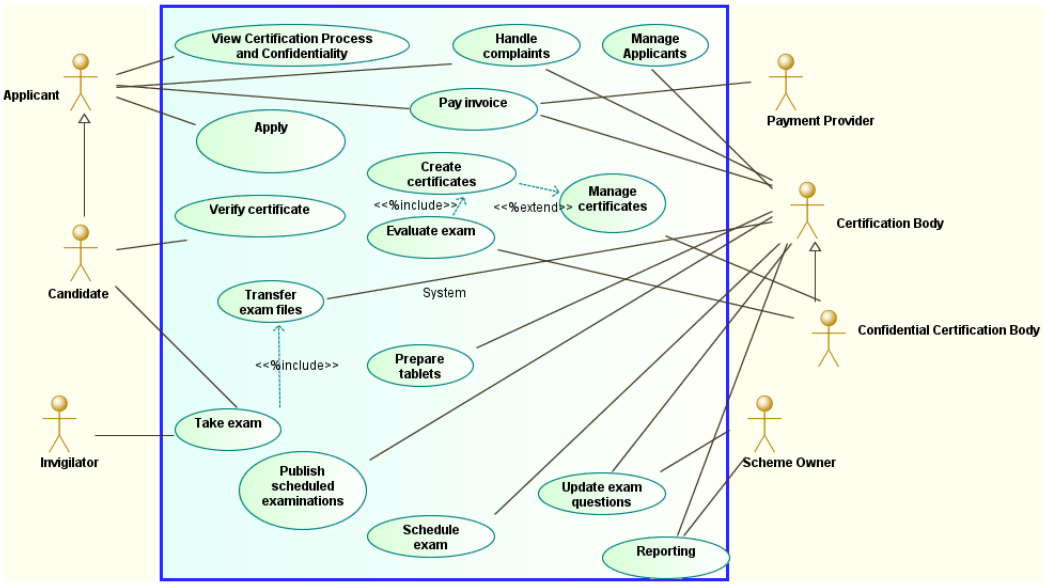

Abbildung 13: Aus den mit dem/der KundIn ermittelten Usecases wird ein Usecasediagramm erstellt

6.1.1 Erweiterte Dokumentation der Usecases

Parallel zur Erstellung des Usecasediagramms werden zusätzliche Parameter und Beschreibungen für jeden Usecase aufgenommen, welche im eigentlichen Diagramm nicht oder nur umständlich zu modellieren, aber wichtig für die Erstellung der Architektur sind. Dafür wird ein Anforderungstemplate, auch Usecasebeschreibung genannt, verwendet [28, S. 214], welches aufbauend auf einer Grundversion [28, Abbildung 8.14, S. 215] erweitert wird und für jeden Usecase folgende Angaben aufnimmt:

- Id: eine eindeutige Bezeichnung, welche verwendet wird, um den Usecase zu referenzieren
- Actor: eine Auflistung aller TeilnehmerInnen des Usecases
- Description: eine kurze Beschreibung des Usecases
- Preconditions: eine Auflistung von Vorbedingungen für den Usecase
- Postconditions: eine Auflistung von Nachbedingungen für den Usecase
- Normal Course of Events: eine Beschreibung des Standardablaufes
- Alternative Courses: Auflistung von Erweiterungen oder zusätzlichen Pfaden des Usecases
- Exceptions: Beschreibung von diversen Ausnahme- und Fehlerfällen
- Assumptions: Annahmen, unter welcher der Usecase beschrieben wird

- Priority: eine Gewichtung, wie wichtig der Usecase ist: Low, Medium oder High
- Related Usecases: eine Auflistung von verwandten Usecases
- Notes: sonstige Anmerkungen

Sind die Abläufe komplexer, können Aktivitätsdiagramme verwendet werden, um Diese wie zB. in Abbildung 14 verständlicher darzustellen [28, S. 215]:

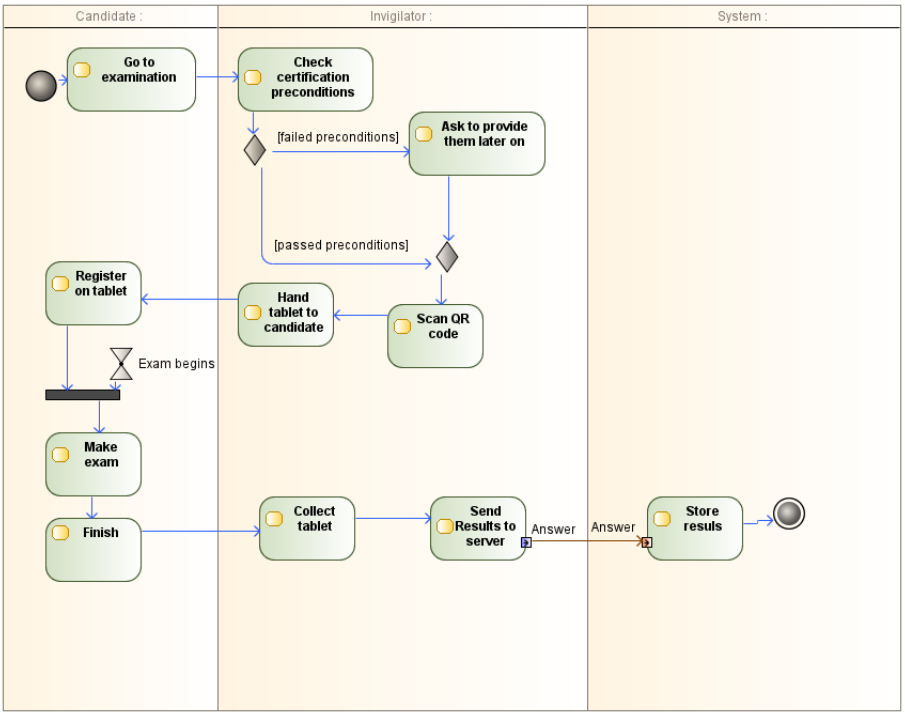

Abbildung 14: Der Ablauf des Take Exam Usecases im Detail

6.1.2 Einbeziehen von Architekturreviewparametern

Um die Einhaltung der Qualitätsparameter zu garantieren wird empfohlen, dass Architekturreviews durchgeführt werden [20, S. 20]. Es existiert zwar „keine singuläre, allgemein akzeptierte Metrik um eine Architektur zu beurteilen“[20, S. 19], jedoch liefern sie grobe Einschätzungen über die Angemessenheit des Systems [20, S. 20]. Folgende Architekturreviews wurden dafür ausgewählt:

- ATAM: betrachtet Wachstums- und explorative Szenarien um die Architektur zu beurteilen [20, S. 61].
- CBAM: basiert auf ATAM, beachtet jedoch vor allem den Nutzen, die Risiken und Kosten der Architektur, um die Architekturentscheidungen besser abwägen zu können. Hauptfaktor ist der ROI. [20, S. 67]

Für diese Reviews können bereits früh ein Großteil der benötigten Parameter ermittelt bzw. zumindest grob abgeschätzt werden. Dies ist wichtig, weil nach der Zehner-Regel der Fehlerkosten früh erkannte Fehler und Probleme weniger Kosten nach sich ziehen als später Erkannte [25, S. 154].

Deswegen wird das Anforderungstemplate um folgende Parameter erweitert:

- Earned Value per Month: Wie viel Umsatz der Usecase in einer bestimmten Zeit generiert
- Expected Usage: Anzahl der erwarteten NutzerInnen des Systems pro Zeiteinheit
- Growth Scenarios: Anzahl der erwarteten NutzerInnen des Systems pro Zeiteinheit bei einer höheren NutzerInnenanzahl und der erwartete Umsatz durch eine Steigerung dieser NutzerInnenzahlen
- Change Scenarios: mögliche Änderungsszenarien und Erweiterungen

6.1.3 Einbeziehen von überprüfbaren, nicht funktionalen Qualitätsattributen

Nicht funktionale Qualitätsattribute beschreiben die nicht funktionalen Anforderungen an das System. Da diese Qualitäten oft ungenau formuliert sind, ist es wichtig, diese Attribute in einer messbaren Form aufzunehmen [32, S. 9].

Deswegen werden für jeden Usecase und dessen architekturrelevanten, nicht funktionalen Anforderungen, messbare Parameter definiert. Diese Parameter sind die Ausgangsbasis für die später folgende Architekturüberprüfung.

Für das Beispielprojekt wurden folgende Parameter definiert, mit welchen das Anforderungstemplate erweitert wurde:

- Response Time in Seconds: Wie schnell die Antwort des Systems auf eine Anfrage reagieren muss. Wichtig ist hier die Unterscheidung in zwei Kategorien: ist die Anforderung verbindend oder nur eine Empfehlung.

Diese Parameter können von Projekt zu Projekt unterschiedlich sein, je nachdem welche zusätzlichen Einschränkungen vom/von der KundIn ermittelt worden wurden.

6.1.4 Ermittlung der Nachbarsysteme und deren Schnittstellen

Für die Planung der Schnittstellen des Systems sind die Anforderungen und Schnittstellen der Nachbarsysteme von wesentlicher Bedeutung. Die Nachbarsysteme können die Architektur des Systems wesentlich beeinflussen. Ein Beispiel hierfür ist zB. die Frage, ob das Payment System wie in diesem Falle nur eine Schnittstelle zur Abfrage der eingegangenen Zahlungen aufweist, oder ob es ein System nach einer eingegangenen Zahlung selbst verständigen kann. Ersteres erfordert eine kontinuierliche Abfrage der Zahlungen, während Zweiteres eine eigens dafür ausgelegt Schnittstelle zur Benachrichtigung der Zahlungen benötigt.

6.2 Rahmenbedingungen

Zusätzlich zu funktionalen und nicht funktionalen Anforderungen werden auch die Rahmenbedingungen ermittelt, unter welchem das System erstellt werden soll. Diese Anforderungen beinhalten meist den organisatorischen und zeitlichen Ablauf des Projektes und können auch gewisse Technologien vorschreiben, zB. wenn das System in ein bereits bestehendes System integriert werden soll. [20, S. 9][33, S. 110]

Die Rahmenbedingungen des Beispielprojektes lassen sich zum Großteil aus dem ISO Standard für Personenzertifizierungsstellen ermitteln [12] und geben Einsicht in die Vertraulichkeit der Daten. Daraus eröffnen sich weitere Usecases. Sofern möglich werden diese Parameter in das Usecasediagramm und das Klassendiagramm der zu verwendeten Daten mit einbezogen. Auf zeitliche und technologische Rahmenbedingungen wurde im Beispielprojekt nicht eingegangen.

6.3 Ermittlung der Daten

Die verwendeten Daten werden ermittelt und mit Hilfe eines Klassendiagramms modelliert. Dies ist nicht nur wichtig und nützlich, um einen Überblick über die Parameter der zu erstellenden Interfaces zu erhalten, sondern wird später auch einen wesentlichen Beitrag zur Aufteilung des Systems in Komponenten leisten. Im Falle des Beispielprojekts wurden folgende Daten ermittelt und modelliert:

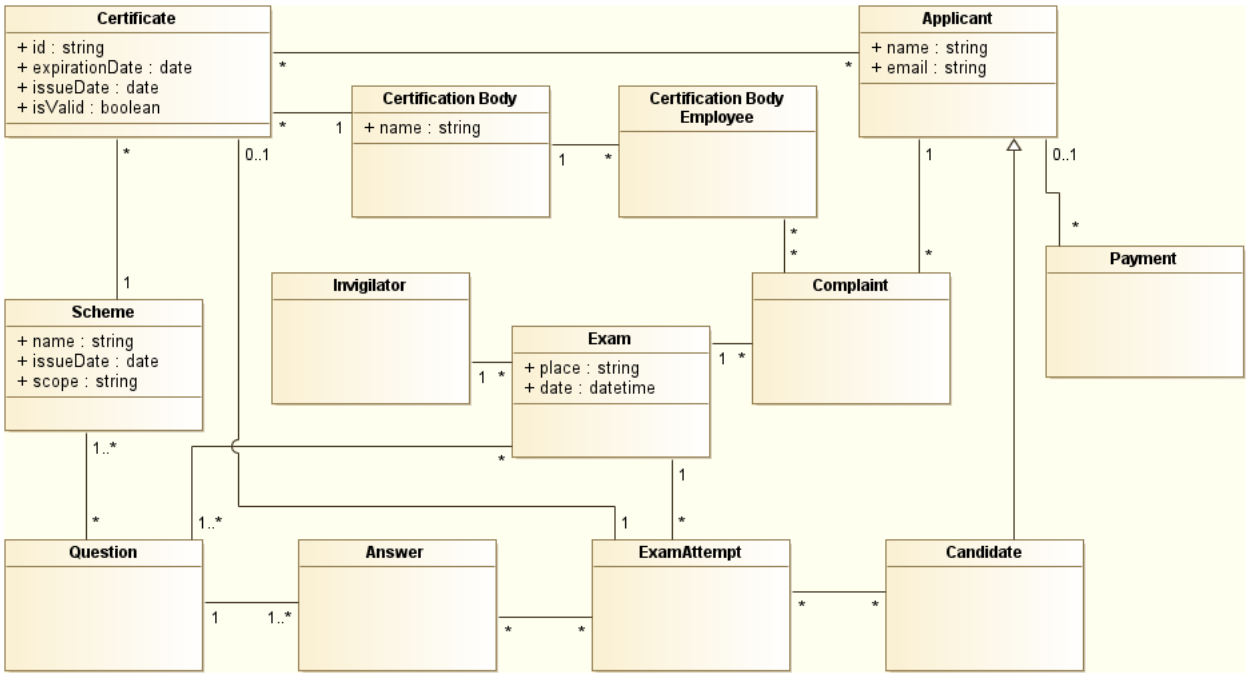

Abbildung 15: Das ermittelte Klassendiagramm des Beispielprojektes

6.4 Ermittlung der Netzwerke

Nach der Ermittlung der Daten wird auf Basis der Rahmenbedingungen und Sicherheitsstruktur zusammen mit dem/der Kunden/Kundin ermittelt, welche Netzwerke für das System benötigt werden. Netzwerke stellen eigene abgeschlossene Bereiche dar, in welchen das System operiert oder von welchem auf das System zugegriffen werden kann. Grundsätzlich existiert mindestens ein Netzwerk. In diesem Fall beherbergt dieses Netzwerk das interne System des Unternehmens.

Soll von anderen Systemen, zB. vom Internet auf Funktionalitäten des Systems zugegriffen werden können, wird ein weiteres Netzwerk benötigt.

Aus den Usecases des Beispielprojektes lässt sich ermitteln, dass in diesem Falle mindestens zwei Netzwerke benötigt werden:

- Public: Usecases die vom Internet auf das System zugreifen, zB. wenn sich ein/eine AnwärterIn (Applicant) für eine Prüfung registriert
- Internal: Usecases, welche nicht vom Internet aus zugänglich sind und interne Verwaltungsaufgaben beschreiben, zB. das Auswerten der Prüfungen

Die Netzwerke werden nun mit einer Vertrautheitsebene von 1 bis n versehen, um eine Sicherheitshirarchie der Netzwerke zu generieren. Es kann vorkommen, dass mehrere Netzwerke mit der selben Vertrautheitsebene versehen werden; diese dürfen sich jedoch nicht überlappen. Zum Beispiel könnte eine weitere Niederlassung der Firma ein eigenes System erstellen, auf welches auch vom Internet aus zugegriffen werden kann. Das Internet würde dann mit der Vertrautheitsebene 1 versehen und beide Systeme mit der Ebene 2. Überlappen sich diese Systeme, muss das überlappende Netzwerk in ein eigenes System ausgegliedert werden.

Die ermittelten Daten werden nun den Netzwerken zugeteilt, für welche sie essentiell zum Betrieb sind. Auch die AkteurInnen werden in Netzwerke aufgeteilt, in welchen sie operieren. Scheinen Daten oder AkteurInnen in mehreren Netzwerken auf, werden sie dem Netzwerk mit der höchsten Sicherheitsebene zugeteilt. Die niedrigeren Bereiche können dann auf diese Bereiche anhand von festgelegten APIs zugreifen.

Im Falle des Beispielprojektes werden alle aufgenommen Daten im Internal Netzwerk verwaltet, da alle Daten als businesskritisch für das Internal System angesehen werden, welches wiederum die höchste Sicherheitsebene besitzt. Würde das Beispielprojekt zB. um ein Forum oder einen Blog erweitert, würden diese dem Public Netzwerk zugeteilt werden; AkteurInnen des Internal Systems können zwar auch auf diese Forum zugreifen oder Blogeinträge schreiben, jedoch sind diese Aktivitäten und Daten nicht kritisch für den normalen Betrieb des Internal Systems. Würde das Forum oder der Blog gehackt werden, resultiert dies zwar in einem Schaden für das Unternehmen, das Internal System könnte aber auch weiterhin normal operiert werden.

Nach der generellen Aufteilung der Daten wird eine Analyse der Rahmenbedingungen durch geführt, um zu erfahren, ob bestimmte Daten aufgrund von Richtlinien besonders geschützt,

und somit in eine eigenes Netzwerk mit einer höheren Vertrautheitsebene ausgelagert werden müssen. Das Beispielprojekt verlangt die vertrauliche Verarbeitung von Prüfungsergebnissen und erwähnt auch explizit das Personal der Prüfungsstelle [12, 7.3]. Weil das Internal Netzwerk, in welchem das Personal operiert, das Netzwerk mit der höchsten Vertrautheitsebene darstellt, muss ein weiteres Netzwerk mit einer höheren Vertrautheitsebene erstellt werden. Dieses Netzwerk wird in diesem Falle mit der Vertrautheitsebene 3 versehen und unter der Beschreibung Confidential geführt.

Um diese Netzwerke besser zu visualisieren zu können, wird das UML Metamodell mit Hilfe eines Profiles angepasst. Jede Netzwerk erhält einen gleich lautenden Stereotypen [29, S. 518]:

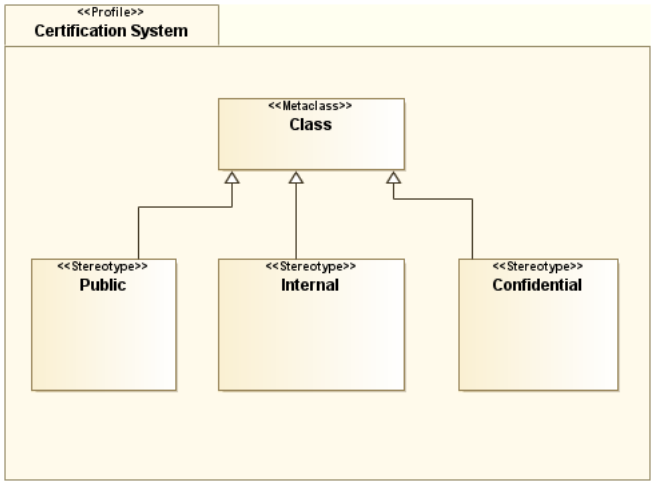

Abbildung 16: Das Metamodell wird mit einem Profil um drei Stereotypen erweitert, welche die Netzwerke der Applikation darstellen

Zusätzlich werden diese Netzwerke auch mit Vertrautheitsebenen verbunden. Diese Ebenen werden im Profil mit einer Notiz versehen, welche die Ebene angibt.

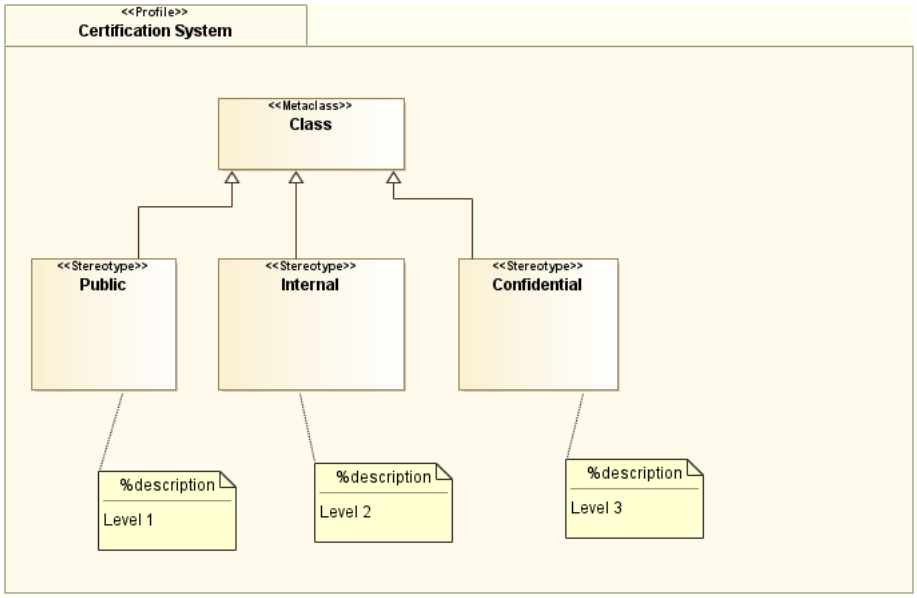

Abbildung 17: Die Netzwerke werden mit Vertrautheitsebenen versehen

Sind die Stereotypen erstellt und mit Vertrautheitsebenen versehen, kann nun damit begonnen werden, die AkteurInnen des Usecasediagramms und die Daten des Klassendiagramms mit diesen Stereotypen zu versehen.

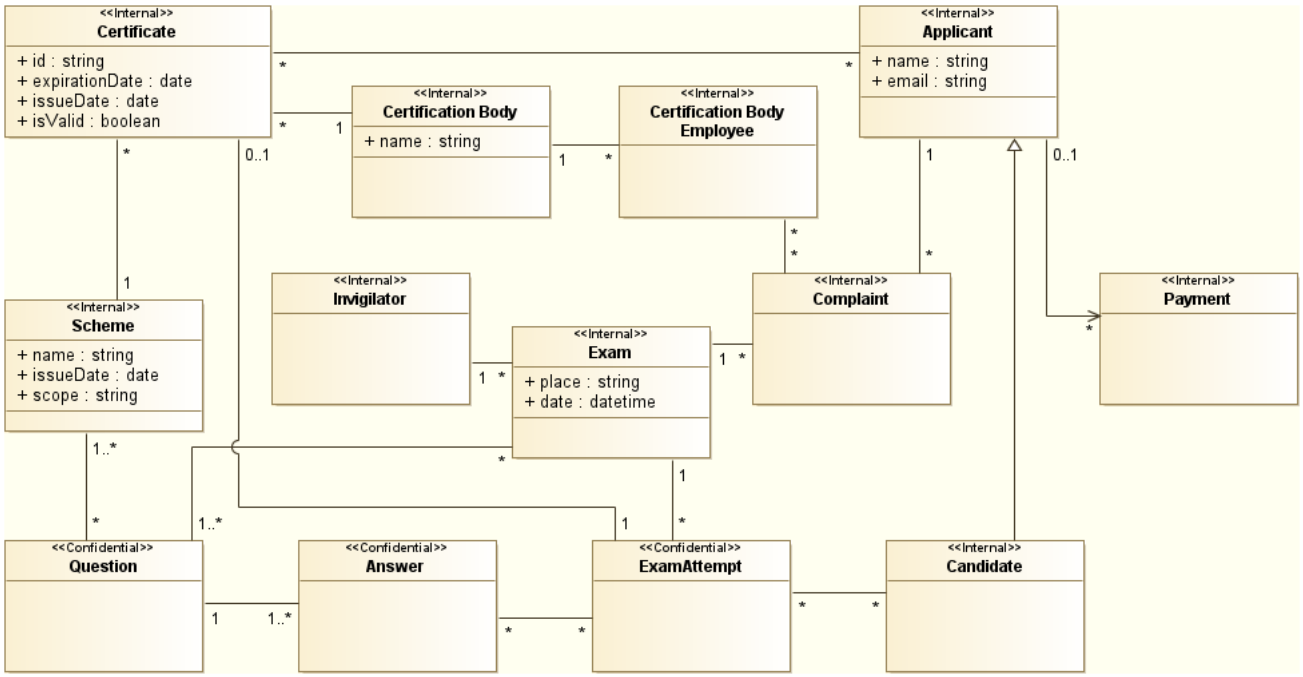

Abbildung 18: Das Klassendiagramm wird mit Stereotypen der Vertraulichkeit erweitert

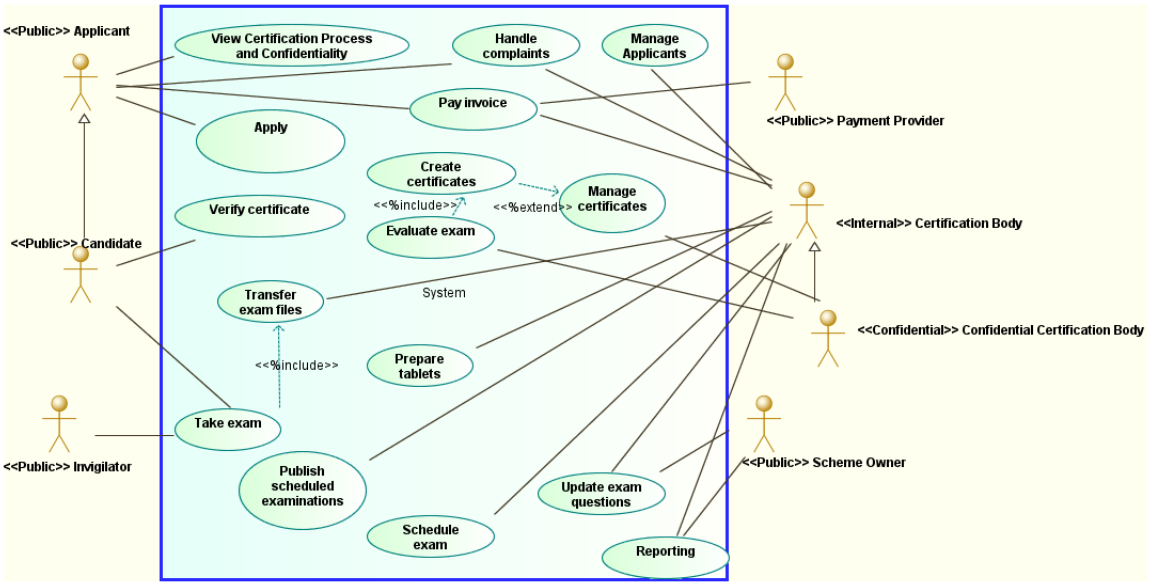

Abbildung 19: Die AkteurInnen des Usecasediagramms wird mit Stereotypen der Vertraulichkeit erweitert

6.5 Ermittlung der Beziehungen zwischen AkteurInnen, Partnersystemen und Daten

Auf Basis des Usecasediagramms können die AkteurInnen und deren Partnersysteme mit Hilfe eines Kontextdiagramms visualisiert werden. Im Gegensatz zum Usecasediagramm geht das Kontextdiagramm auf die zwischen den Systemen und AkteurInnen fließenden Daten ein und stellt so die Verbindungen zwischen Daten, Systemen und NutzerInnen auf.

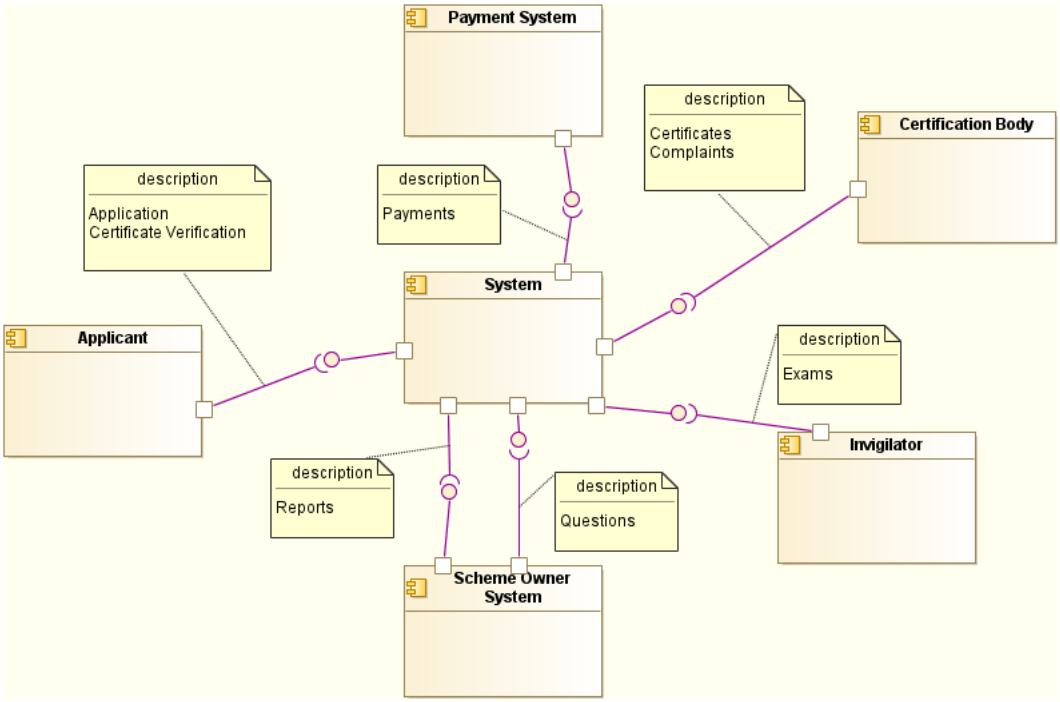

Abbildung 20: Das Kontextdiagramm zeigt das System, die AkteurInnen, die Nachbarsysteme und die dazwischen fließenden Daten

7 Erstellung der Architektur

Aufbauend auf den im Anforderungsprozess ermittelten Attribute kann nun mit der Architekturplanung begonnen werden. Bereits an dieser Stelle kann aufgrund der ermittelten Parameter und Erfahrungswerte eine grundsätzliche Überprüfung der Machbarkeit des Projektes durchgeführt werden. Eine weitere Überprüfung, nämlich ob der beschriebene Prozess sich für die Architekturplanung des Projektes eignet, kann ebenso durchgeführt werden: Der Prozess beschäftigt sich hauptsächlich mit der Aufspaltung und Trennung der Daten und AkteurInnen in mehrere Systeme. Durch außergewöhnlich strenge Laufzeitanforderungen oder entsprechende Rahmenbedingungen kann diese Aufspaltung jedoch zu einer Architektur führen, welche die ursprünglich ermittelten Anforderungen nicht mehr, oder nur schlecht erfüllt.

Die erwähnten Architektursichten werden, soweit möglich, durch UML Diagramme realisiert:

- Logical View: Klassendiagramm
- Process View: Komponentendiagramm, diverse Werte in der Usecasebeschreibung sowie Aktivitäts- und Interface-Klassendiagramm.

- Development View: Da noch keine Implementation vorhanden ist, können noch keine Bibliotheken und Module beschrieben werden. Hierfür würde sich jedoch das Paketdiagramm anbieten.
- Physical View: Grobes Komponentendiagramm. Da die Wahl der exakten physischen Komponenten durch die fehlende Implementation nicht früh überprüfbar ist, wurde auf eine Modellierung in der Planungsphase verzichtet.
- Scenarios: Usecasediagramm.

7.1 Erstellen der minimalen Architektur

Das Kontextdiagramm, welches im Anforderungsprozess erstellt worden ist, zeigt das System mit allen AkteurInnen und Nachbarsystemen. Aufbauend darauf kann nun die minimale Architektur erstellt werden, welche sich aus dem System und den Nachbarsystemen ableitet.

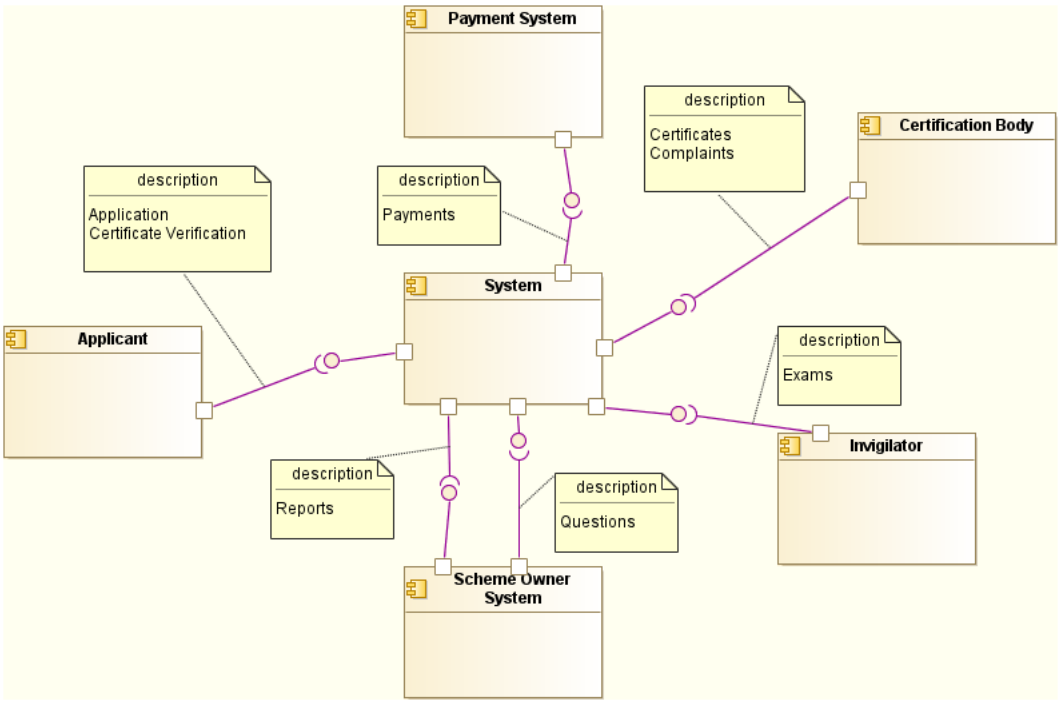

Abbildung 21: Das Kontextdiagramm liefert die Ausgangsbasis für die Architektur

Zuerst werden alle Datenflussnotizen entfernt. Danach werden alle Komponenten entfernt, welche kein eigenes System darstellen. In diesem Falle werden folgende Komponenten entfernt:

- Applicant
- Certification Body

- Invigilator

Dies führt zu folgender Minimalarchitektur:

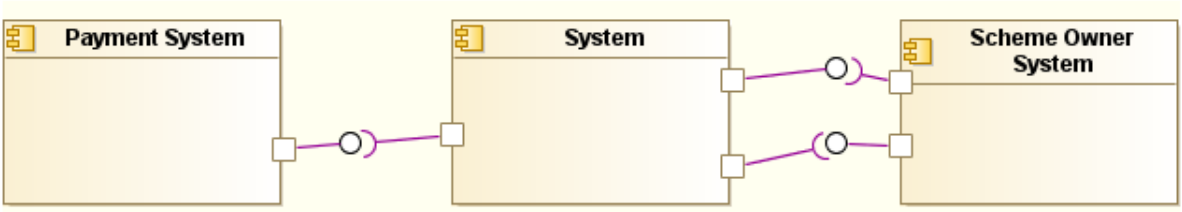

Abbildung 22: Minimale Architektur

Für die Nachbarsysteme selbst wird keine Architektur erstellt, jedoch beeinflussen sie die Schnittstellen des Systems und sind deswegen wichtig für den weiteren Prozess. Sie werden daher in die Architekturplanung einbezogen.

7.2 Erstellen der Datenminimalarchitektur

Auf Basis der im Anforderungsprozess ermittelten Netzwerke wird die vorher erstellte Minimalarchitektur in ebenso viele Teilsysteme unterteilt. Die Aktivitätsdiagramme werden an die neue Architektur angepasst: Für jedes Untersystem wird in den Diagrammen eine eigene Swimlane erstellt. Die involvierten AkteurInnen sind, falls möglich, als eigene Swimlane modelliert, spielen in dieser Phase aber noch keine wichtige Rolle zur Gliederung des Systems.

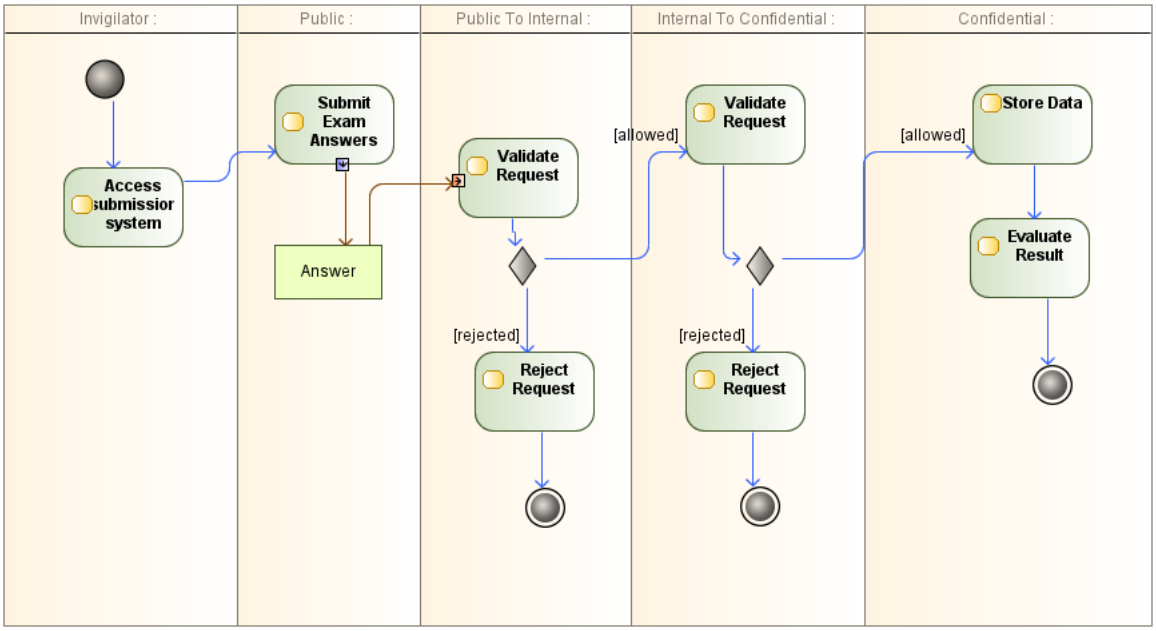

Abbildung 23: Die Antworten werden nach der Prüfung an den Certification Body übermittelt. Der Request wird dann durch zwei Gateways zum finalen System geleitet.

Wechselt der Kontrollfluss eine Swimlane eines Systems, heißt dies, dass eine Verbindung zwischen den beiden sonst abgeschotteten Systemen benötigt wird. Dieses Verbindung wird als eigene Komponente modelliert und wird als Gateway bezeichnet. Die Aufgabe dieses Gateways ist es, folgende Attribute der Anfrage zu überprüfen und die Anfrage gegebenenfalls zu verwerfen oder weiterzuleiten:

- Von welchem System kommt die Anfrage?
- Welches System ist das Ziel der Anfrage?
- Welche Schnittstelle dieses Systems ist das Ziel der Anfrage?
- Gibt es eine Regel die diese Anfrage explizit erlaubt?

Der Gateway fungiert damit als eine Art Application Firewall.

Der Gateway ist jedoch nicht der einzige Punkt an welchem der Zugriff auf Schnittstellen autorisiert wird. Er ist viel mehr ein zusätzlicher Layer, der den Zugriff auf die Schnittstellen absichert. Überwindet ein/eine AngreiferIn durch einen Fehler im Gateway diese Kontrolle, hat er/sie zwar Zugriff zum Internal System, kann aber trotzdem noch nicht auf die Daten zugreifen, da diese eine zusätzliche Authentifizierung benötigen. [23][S. 350]

Die anfangs beschriebenen Nachbarsysteme werden nach ihren Anforderungen, welche aus den Aktivitätsdiagrammen ablesbar sind, an das System in ihrem Netzwerk angeschlossen. Ist das Ausgangssystem ein System, welches nicht vom/von der AuftraggeberIn kontrolliert wird,

und greift das Ausgangssystem von einem Netzwerk mit einer niedrigeren Vertrautheitsebene auf ein Zielsystem mit einer höheren Vertrautheitsebene zu, muss dies jedoch über ein zusätzliches System erfolgen. Dies ermöglicht es, den Gateway komplett von unkontrollierten Systemen abzukapseln und dessen Angriffsfläche zu verringern. Dies ist besonders wichtig, weil der Gateway bei Angriffen einen Single Point of Failure darstellt.

Das Beispielprojekt bezieht Zahlungsdaten direkt von einem Payment System und die Prüfungsfragen werden direkt an das Scheme Owner System gesandt. Beides Ausgangssysteme dieser Anfragen stammen aus einem System mit einer höheren Vertrautheitsebene als dem Zielsystem und benötigen deswegen kein eigenes Zwischensystem, sprich können direkt über den Gateway zum Ziel geführt werden. Dem entgegen gesetzt ist die Übermittlung der Prüfungsfragen des Scheme Owner Systems: Das Ausgangssystem greift hier von einem nicht kontrollierten System und einer niedrigeren Vertrautheitsebene auf eine Zielsystem in einer höheren Vertrautheitsebene zu, was eine zusätzliche Komponente nötig macht.

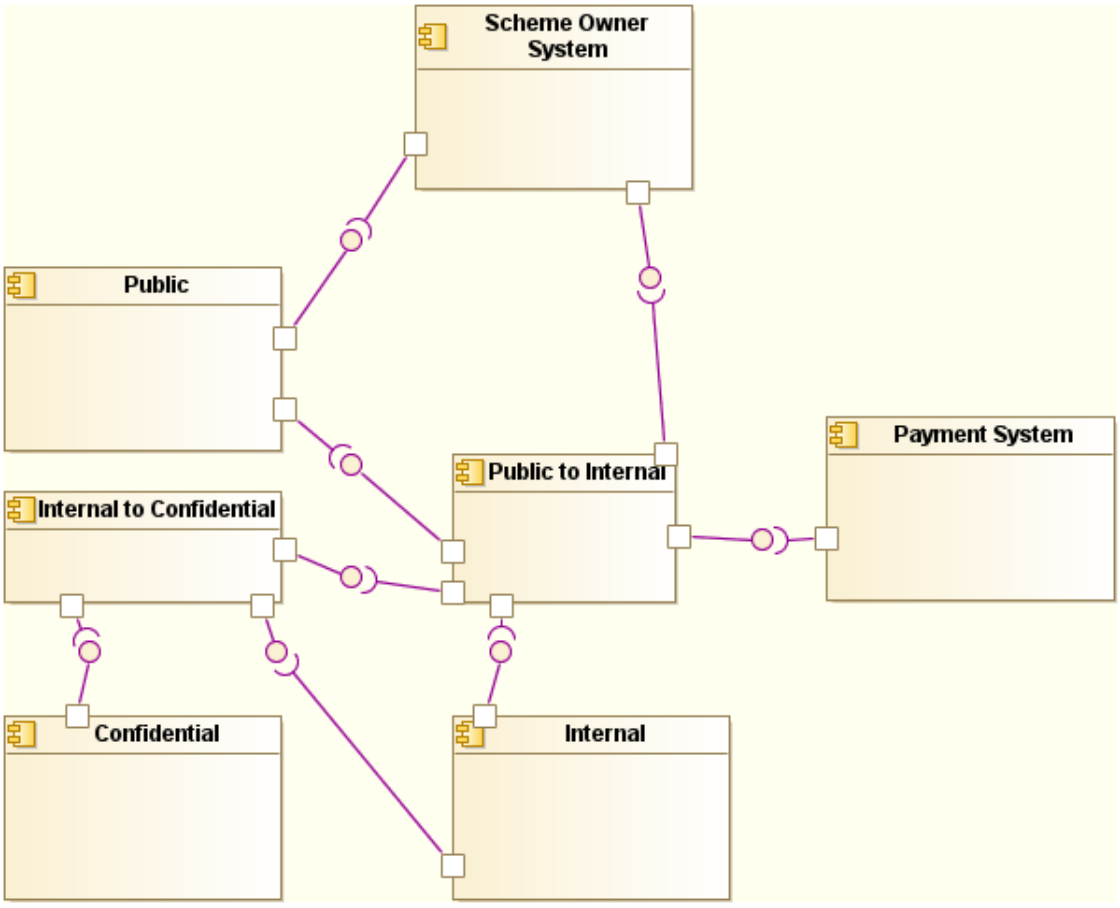

Abbildung 24: Aufteilung der Komponenten in Datenbereiche

Eine weitere wichtige Regel ist, dass keine Gateways unterschiedlicher Vertrautheitsebenen übersprungen werden dürfen. Zeigt ein Aktivitätsdiagramm zB. einen Zugriff von Ebene 1 auf Ebene 3 muss dieser Zugriff sowohl durch den Gateway der Ebene 2, als auch durch den Ga-

teway der Ebene 3 geleitet werden. Dies verhindert, dass besonders schützenswerte Systeme direkt an Systeme mit einer weitaus niedrigeren Vertrautheitsebene angeschlossen werden und so dessen Gateway zum Single Point of Failure wird. Dies gilt in beide Richtungen.

Da bei der Erstellung des Systems nun alle Schnittstellen und Systeme bekannt sind, können diese Regeln fest im Gateway verankert werden. Weil diese Gateways unabhängig voneinander agieren, können sie durch das Hinzufügen eines Load Balancers beliebig vervielfacht werden, was sowohl die Ausfallicherheit als auch die Skalierbarkeit erhöht. Das ist wichtig, weil sie als einzige Verbindung zwischen den Systemen zu einer Art Flaschenhals werden.

7.3 Einbinden der AkteurInnen

Nachdem die Datenminimalarchitektur steht, können nun die AkteurInnen des Systems in die Aufgliederung des Systems mit einbezogen werden. Hierfür müssen nun die Objektflüsse und die AkteurInnen des Systems für jeden Usecase betrachtet werden, welche aus den vorher bereits erstellten Aktivitäts- und Kontextdiagramm ersichtlich sind.

Zuerst wird das erste Untersystem, in diesem Falle das Public System, betrachtet. Alle Objektflüsse durch das System und die AkteurInnen, welche mit ihren Swimlanes angrenzen, sind in die Aktivitäten des Systems involviert. Jede Involvierung eines/einer Akteurs/Akteurin in ein System erfordert einen Zugang zu diesem System.

Jeder dieser AkteurInnen muss mit den minimal möglichen Rechten für dieses System ausgestattet werden, um seine Aufgaben zu erfüllen. Dies vermeidet nicht nur Fehler sondern reduziert auch den Schaden, welcher ein potentieller Angriff dieses Akteurs/dieser Akteurin anrichten kann [14, 1. A].

Da ein System komplex ist [33, S. 7] und diese Sicherheitsattribute nach Änderungen am System immer wieder überprüft werden müssen, stellt jeder zusätzliche Zugriff eines/einer Akteurs/Akteurin nicht nur ein Sicherheitsrisiko dar, sondern erhöht auch den Test- und damit den Wartungsaufwand. Idealerweise wird daher jedem/jeder AkteurIn ein eigenes, für sich abgekapseltes System zur Verfügung gestellt, was jedoch meist aufgrund Kosten der zusätzlichen Systeme keine Option darstellt.

Um zu ermitteln, welche Systeme eine eigene Komponente benötigen, wird nun entweder anhand einer Tabelle oder zusammen mit dem/der KundIn pro Usecase und deren Komponenten ermittelt, ob der Schaden eines unerlaubten Zugriffs der Daten den eines separaten Systems überschreitet. Die Schadens- und Systemkosten müssen zuerst von dem/der KundIn und dem/der ArchitektIn ermittelt bzw. geschätzt werden.

Im Falle des Beispielprojektes wurde auf Basis des folgenden stark vereinfachten Aktivitätsdiagramms in Abbildung 25 ermittelt, dass die möglichen Schadenskosten im Falle, dass der/die AnwärterIn (Applicant) Zugriff auf die Prüfungsantworten (Answer) bekommt, die eines eigenen Systems überschreiten. Das gleiche Problem trifft auch auf den Scheme Owner zu: die Schadenskosten im Falle einer Manipulation oder eines lesenden Zugriffes des/der AnwärterIn (Applicant) auf die Fragen überschreitet auch hier die Kosten eines eigenen Systems. Deswegen

werden zwei zusätzliche Systeme erstellt und aus dem Public System ausgegliedert.

Für den Fall, dass bei der Aufspaltung zu viele Systeme entstanden sind, werden nun in einem weiteren Schritt diverse Kombinationen von Teilsystemen betrachtet und versucht zusammen zu legen, solange deren Schadenskosten nicht die Systemkosten überschreiten. Gibt es mehrere mögliche Kombinationen, entscheidet das in den Anforderungen aufgenommene Related Usecases Feld und danach die Überschneidung der Datentypen über die genaue Aufteilung. Ändert sich ein Usecase, so sind meist auch andere Usecases von dieser Änderung betroffen. Je weniger Komponenten nach einer Änderung betrachtet werden müssen, desto niedriger sind die Wartungskosten.

Beim Beispielprojekt zeigt sich, dass es keine erlaubte Kombination gibt, da die Schadenskosten jeweils weit über den Kosten eines eigenen Systemes liegen. Es bleibt somit bei den ermittelten zwei Zusatzsystemen.

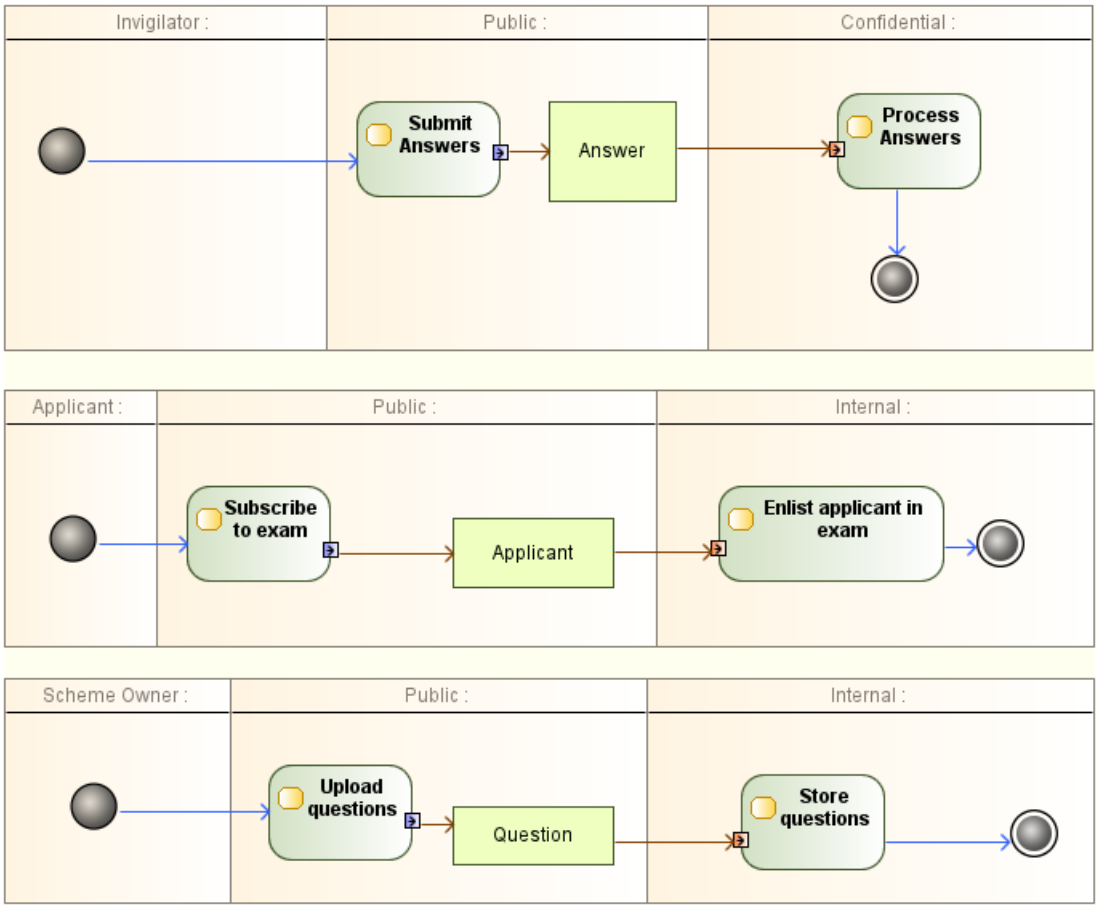

Abbildung 25: Vereinfachte Gegenüberstellung von Aktivitätsdiagramme für das Public System

Diese Analyse wird für alle verbleibenden Systeme durchgeführt, bis alle Systeme aufgespalten sind.

Im Falle des Beispielprojektes führt dies schlussendlich zu folgender Systemaufspaltung:

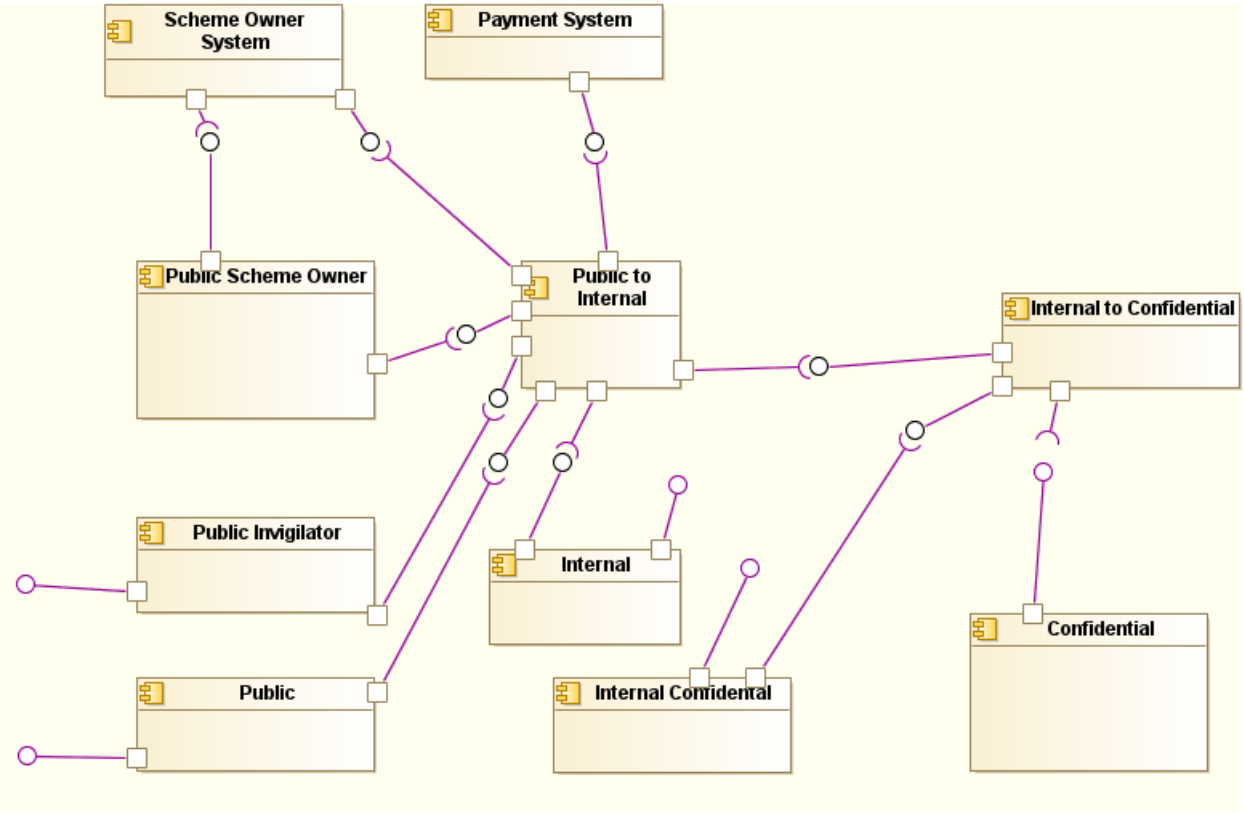

Abbildung 26: Architektur nach der Aufspaltung

7.4 Analyse der nicht funktionalen Attribute

Ist die erste Version der Architektur erstellt, kann nun mit der grundsätzlichen Überprüfung der im Anforderungsprozess ermittelten Parameter begonnen werden, welche bereits wichtige Informationen und Rückschlüsse auf den jetzigen Status der Architektur geben. Basierend auf den Usecases, den ermittelten Komponenten der Architektur und den Aktivitätsdiagrammen wird eine Tabelle erstellt, welche Auskunft darüber gibt, welche Komponenten für jeden Usecase benötigt werden. Diese Tabelle dient als Basis für weitere Analysen.

Um herauszufinden, welche Komponenten für einen Usecase benötigt werden, können die Swimlanes der Aktivitätsdiagramme herangezogen werden. Da für AkteurInnen keine Komponente gelistet werden, werden deren Swimlanes ignoriert. Ein Beispiel hierfür ist das Aktivitätsdiagramm des Handle Complaint Usecases in Abbildung 27.

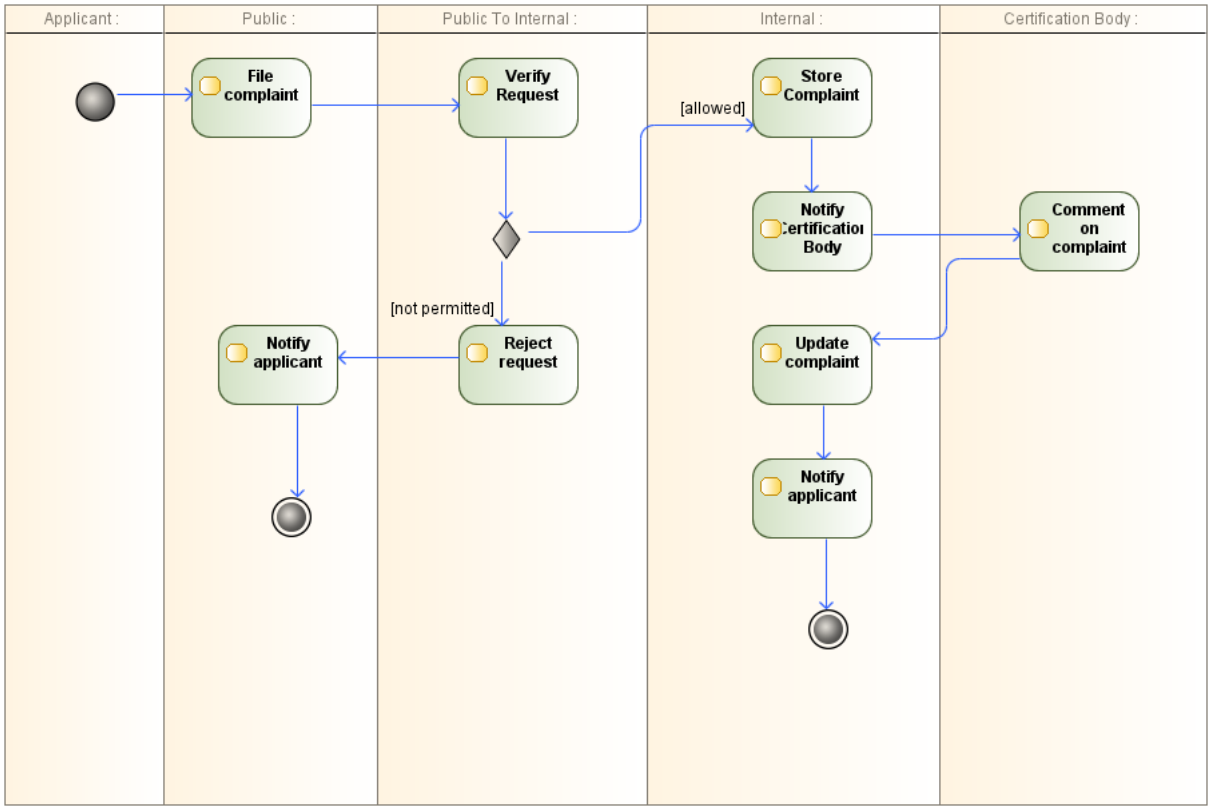

Abbildung 27: Vereinfachtes Aktivitätsdiagramm des Handle Complaints Usecases mit der id complaints

Werden die beiden AkteurInnen entfernt, bleiben für diesen Usecase folgende Komponenten übrig:

- Public
- Public To Internal
- Internal

Diese werden nun in eine Tabelle überführt, wobei für jede verwendete Komponente eines Usecases die entsprechende Zelle mit einem x markiert wird:

Komponenten	complaints
Public	x
Public Invigilator	
Public Scheme Owner	
Public To Internal	x
Internal	x
Internal Confidential	
Internal To Confidential	
Confidential	
Scheme Owner System	
Payment System	

Dies wird für alle Usecases durchgeführt und führt im Falle des Beispielprojektes zu folgender Matrix:

	Public	Public Invigilator	Public Scheme Owner	Public To Internal	Internal	Internal Confidential	Internal To Confidential	Confidential	Payment System	Scheme Owner System
pay-invoice				x	x				x	
apply	x			x	x					
verify-cert	x			x	x					
take-exam		x		x			x	x		
publish-exams	x			x	x					
schedule-exam	x			x	x					
reporting				x	x					x
evaluate-exam						x	x	x		
update-questions			x	x		x	x	x		
prepare-exam					x					
complaints	x			x	x					
manage-applicants					x					
manage-certs					x					
meta-information	x			x	x					

Abbildung 28: Matrix der Komponenten und Usecases des Beispielprojektes

7.4.1 Reliability

Anhand der in ermittelten Usecase und Komponenten Matrix kann nun sowohl eine Überprüfung der der Ausfallkosten als auch eine Single Point of Failure Analyse durchgeführt werden.

Single Point of Failure Analyse

Ein Single Point of Failure beschreibt eine Komponente, die so kritisch für das System ist, dass ihr Ausfall den kompletten Ausfall des Systems nach sich zieht [18, S. 3]. Ein Single Point of Failure der Architektur kann daran erkannt werden, dass eine Komponente in allen Usecases vorkommt. Im Bezug auf die soeben ermittelte Matrix wird dies durch eine durchgehende Reihe von mit x markierten Zellen in der Komponentenspalte ersichtlich.

Enthält eine Architektur einen Single Point of Failure, muss diese Komponente entweder redundant ausgelegt sein, oder es muss eine Aufspaltung anhand einer Fehlerkostenanalyse

durchgeführt werden.

Im Falle der Matrix in Abbildung 28 ist keine durchgehende Reihe an markierten Zellen erkennbar und somit existiert kein Single of Failure. Es lässt sich einzig und allein ablesen, dass die Public To Internal und Internal Komponente eine Abweichung vom Status eines Single Point of Failures um vier respektive drei Punkte besitzen. Dies lässt erahnen, dass bei der Implementation und Wartung dieser Systeme besondere Sorgfalt von Nöten ist.

Ausfallkosten Analyse

Zusätzlich zu einer Single Point of Failure Analyse wird eine Ausfallkostenanalyse durchgeführt. Diese basiert auf den in den Anforderungen ermittelten Monatsumsätze jedes Usecases. Für diese Überlegung können auch die Wachstumsszenarien einbezogen werden, falls sich der Umsatz durch eine Steigerung der BenutzerInnen steigert oder senkt.

Das Ergebnis dieser Analyse ist die notwendige Redundanz der einzelnen Komponenten, welche bei der Implementation des Systems erreicht werden muss.

Die nachfolgenden Berechnungen verteilen den Schaden aus Gründen der Einfachheit gleichmäßig auf alle Zeiteinheiten. Es kann sein, dass eine Organisation so strukturiert ist, dass ein eintägiger Ausfall keine Umsatzeinbußen nach sich zieht. In diesem Falle ist die Beispielrechung ungenau. Um diese Möglichkeit einzubeziehen, müssen detailiertere Umsatzkosten im Anforderungsprozess ermittelt werden was jedoch aus Gründen des Umfangs und der Einfachheit vermieden wurde. Das Beispiel soll lediglich als ein anschauliches Beispiel zur Ermittlung der erforderlichen Ausfallsicherheit dienen.

Um den monatlichen Umsatz der Komponenten zu ermitteln, wird das x in den markierten Zellen mit dem Monatsumsatz des Usecases ersetzt. Leere Felder werden mit 0 aufgefüllt. Schlussendlich werden alle Werte einer Komponente addiert, um den Gesamtumsatz der Komponente zu ermitteln.

Als Beispiel dient hier die Internal Komponente (Die Werte sind beispielhaft gewählt):

Usecases	Internal
pay-invoice	3000 €
apply	1500 €
verify-cert	1500 €
take-exam	0 €
publish-exams	1500 €
schedule-exam	1500 €
reporting	1500 €
evaluate-exam	0 €
update-questions	0 €
prepare-exam	80000 €
complaints	1500 €
manage-applicants	1500 €
manage-certs	1500 €
meta-information	1500 €
Total Sum	96500 €

Fällt diese Komponente für einen kompletten Monat aus, verursacht sie einen Umsatzrückgang von 95500 €. Wenn eine Komponente eine Ausfallwahrscheinlichkeit von 99% besitzt, dann belaufen sich die durchschnittlichen Ausfallkosten auf 965 €. Wird aus Redundanzgründen eine weitere Komponente hinzugefügt und damit die Ausfallwahrscheinlichkeit auf zB. 99.9% gesteigert, belaufen sich die Schadenskosten nur noch auf 96.5 €. Kostet diese zusätzliche Komponente weniger als die Differenz der Kosten, sprich weniger als 868.5 €, rentiert sich die Anschaffung einer zweiten Komponente.

7.4.2 Usability

Da die Artefakte des Planungsprozesses keine Oberflächen beschreiben ist eine Auswertung der des nicht funktionalen Parameters Usability nicht möglich. Außerdem ist sie nicht im Fokus der Architekturerstellung und die Überprüfung der Usability kann somit übersprungen werden.

7.4.3 Efficiency

Es ist äußerst schwierig, die Effizienz und Performance der Architektur in diesem frühen Stadium zu messen, da noch keine Implementation vorhanden ist und somit weder die Performance noch der Speicherverbrauch des Systems getestet werden kann. In diesem Stadium lassen sich lediglich Werte schätzen. Eine Möglichkeit um zB. die Antwortzeiten zu schätzen, ist es, die An-

zahl der Swimlanewechsel eines Usecases zu addieren und das Ergebnis mit einer konstanten Zeiteinheit, welche auf Erfahrungswerten basiert, zu multiplizieren. Dieser Wert kann dann mit den im Anforderungsprozess ermittelten Antwortzeiten verglichen werden, um zu überprüfen, ob das System diese Anforderungen erfüllt.

Ausgegangen wird hier von den bestehenden Aktivitätsdiagrammen. Je nachdem, für welchen Abschnitt die in den Anforderungen ermittelten Werte gelten, kann nicht der komplette Ablauf des Aktivitätsdiagrammes für die Berechnung der Zeit verwendet werden.

Überschreitet der berechnete Wert den in den Anforderungen ermittelten Wert, muss überprüft werden, ob die Einhaltung des Wertes nur eine Empfehlung oder eine Verpflichtung ist. Ist der Wert nur eine Empfehlung, so wird der Usecase in der Implementationsphase mit einem besonderen Augenmerk auf Geschwindigkeit umgesetzt. Ist der Wert jedoch verbindlich, so muss mit dem/der KundIn Rücksprache gehalten werden [32, S. 70]: Entweder ist die Anforderung unter diesen Parametern nicht umsetzbar, oder es muss ein Kompromiss zu lasten von anderen Anforderungen eingegangen werden.

Ein Beispiel für die Berechnung der Antwortzeiten wird aus dem Aktivitätsdiagramm des Beispielprojektes für die Anmeldung eines Kandidaten erläutert (Abbildung 29):

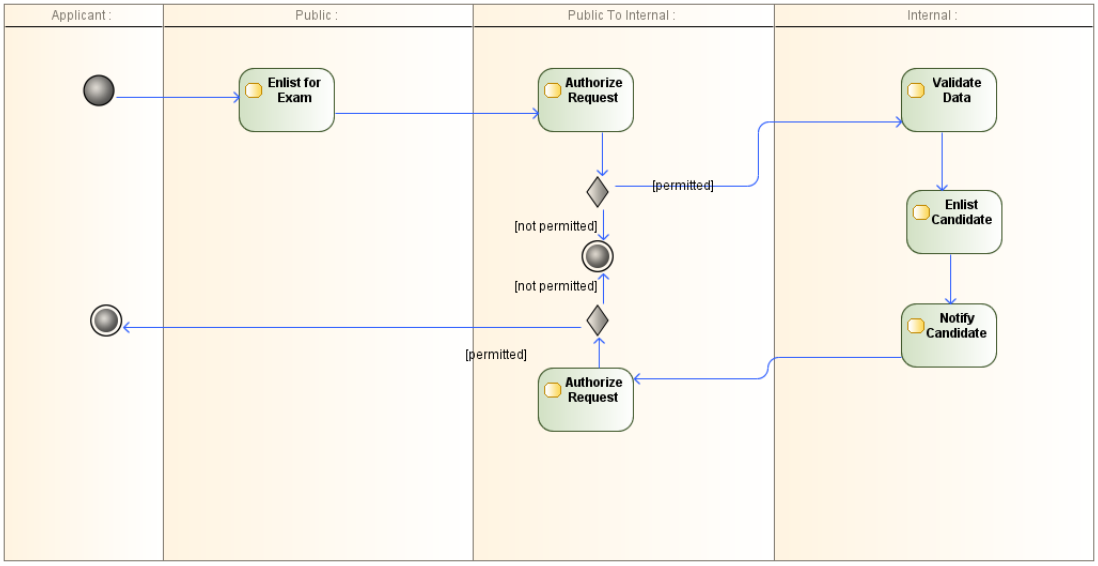

Abbildung 29: Der Kandidat meldet sich für eine Prüfung an

Hier werden, ausgegangen vom Startpunkt sechs Swimlanewechsel gezählt. Diese werden nun mit der Konstante 100 Millisekunden multipliziert, was eine geschätzte Durchlaufzeit von 600 Millisekunden ergibt. Dies liegt unter den erforderlichen 1000 Millisekunden der aufgenommenen nicht funktionalen Anforderung der Antwortzeit.

7.4.4 Maintainability

Maintainability dreht sich um die Wart- und Änderbarkeit eines Projektes. Die Änderbarkeit der Architektur hängt wesentlich von der Kopplung und Kohäsion der Komponenten ab. Die Wartbarkeit kann unter Anderem an den Wartungskosten abgelesen werden.

Kopplung und Kohäsion

Die Änder- und Wartbarkeit ist wesentlich von den Beziehungen der Komponenten untereinander und zu den Usecases abhängig. Wird eine Komponenten geändert, so müssen zusätzlich zu dieser Änderung auch eine erneute Validierung der bereits bestehenden Funktionen durchgeführt werden, um Fehler, welche aus den Änderungen entstanden sind, zu finden. Dieses Problem wird mit dem Begriff Ripple Effect beschrieben. [2, S. 3]

Wird ein Usecase geändert, so müssen eine oder mehrere Komponenten des Usecases angepasst werden. Wird eine Komponenten geändert, so kann dies auch andere Usecases beeinflussen, welche die gleiche Komponente verwenden. Wie sehr eine Komponente andere Usecases beeinflusst, kann aus den Komponenten Spalten der Matrix abgelesen werden, in dem die Anzahl der mit x markierten Spalten durch die Anzahl der Gesamtusecases dividiert wird. Der daraus resultierende Wert liegt in den Grenzen von 0 (bester Wert) und 1 (schlechtester Wert) und gibt die Kopplung der Komponente an. [32, S. 164]

Die Kohäsion beschreibt die inhaltliche Zusammengehörigkeit der Komponenten [32, S. 164]. Sie kann aus den der Usecase-, Aktivitäts und Klassendiagrammen ermittelt werden. Das Klassendiagramm beschreibt die Zusammengehörigkeit der Daten, das Aktivitätsdiagramm eines Usecases und dessen Objektflüsse die gemeinsam verwendeten Daten. Die Usecases können nun anhand der verwendeten Daten in mehrere unabhängige Gruppen unterteilt werden, die orthogonal zueinander sind. Je mehr unabhängige Gruppen auf eine Komponente zugreifen, desto schlechter ist deren Kohäsion.

Zusätzlich können nun die in den Anforderungen ermittelten Änderungsszenarien mit einbezogen werden. Hat eine Komponente einen hohen Kopplungswert und/oder eine niedrigen Kohäsion, und ist sie Teil eines Usecases, welcher viele Änderungsszenarien beinhaltet, kann dies als Grund zu einer weiteren Aufspaltung der Komponente genommen werden. Gateways sind von Aufspaltungen ausgenommen, da sie als Firewall agieren und so simpel wie möglich konfiguriert werden müssen, um Konfigurationsfehler zu vermeiden.

Wird eine Aufspaltung durchgeführt, so werden sie anhand der in Kohäsionsanalyse ermittelten Gruppen aufgeteilt. Die Priorität der Gruppen wiederum ergibt sich aus den ermittelten Änderungsszenarien, danach werden die in der Ausfallkostenanalyse ermittelten Kosten einer Komponente herangezogen. Diese Ordnung ergibt sich aus einer Studie, welche ermittelt hat, dass die Wartung der Software oft mehr als 50% der Gesamtsystemkosten des Systems ausmacht [5, S. 71-84].

Die Änderungsszenarien und die Kopplungswerte stellen schlecht bewertbare Werte dar. Zudem kann eine feste Aufteilung anhand der unabhängigen Gruppen der Kohäsion zu einem Sy-

stem führen, in welchem jeder kleine und unabhängige Usecase ein eigenes System bekommt, was wiederum zu einer zu starken Fragmentierung führen kann. Deshalb kann hier keine starre Regel festgelegt werden, wann ein System aufgespalten werden muss und wann nicht. Diese Entscheidung muss vom/von der ArchitektIn selbst getroffen werden.

Wird das Beispielprojekt auf Kohäsion und Kopplung analysiert, fallen vor allem die Internal und Public To Internal Komponente auf, da sie in mehr als der Hälfte aller Usecases eingesetzt werden (Kopplungswert größer als 0.5). Weil die Public To Internal Komponente ein Gateway ist, kann sie ignoriert werden. Damit verbleibt die Internal Komponente. Wird die Kohäsion der Internal Komponente betrachtet, sticht hervor, dass in dieser Komponente mehrere komplett unabhängige Daten verwendet werden: Zahlungen, Zertifikate, Anmeldungen und Beschwerden sind jeweils eigene, orthogonale Gruppen.

Aufgrund der Größe des Systems und der wenigen Änderungsszenarien wurde gegen eine weitere Aufspaltung entschieden, jedoch wurde fest gestellt, dass die Internal Komponente bei der Erstellung der Testfälle eine höhere Testabdeckungen benötigt.

Wartungskosten

Die Wartungskosten ergeben sich sowohl aus dem Personal, welches für die Wartung der Komponenten angestellt werden muss, als auch aus den Strom- und Reparaturkosten der eingesetzten Systeme. Weil die genauen Komponenten noch nicht implementiert wurden, kann an diesem Moment nur eine Schätzung der Kosten durchgeführt werden.

Eine einfache Möglichkeit zur Schätzung der Kosten kann durch die Wahl einer Konstanten pro System durchgeführt werden. Im Falle des Beispielprojektes wird pro System mit folgenden monatlichen Kosten gerechnet:

- Stromkosten: 10 €
- Reparaturkosten: 20 €
- Personalkosten: 100 €

Die aufsummierten Kosten ergeben Wartungskosten von 130 € pro Monat. Wird dies mit der Anzahl der Systeme multipliziert ergibt sich ein Gesamtwartungskostenaufwand von 8 * 130 €, sprich 1040 € pro Monat.

7.4.5 Portability

Die Portabilität der Platform kann im Moment noch nicht überprüft werden, da noch keine Festlegung des Projektes auf eine Plattform und/oder Technologie existiert. Dies kann erst zu Beginn der Implementationsphase entschieden werden und wird unter Anderem von den in der Anforderungsphase ermittelten Rahmenbedingungen beeinflusst.

7.5 Modellierung der Komponentenschnittstellen

Sind alle Analysen der Architektur abgeschlossen, kann nun anhand der Usecase-, Klassen und Aktivitätsdiagramme damit begonnen werden, die Schnittstellen der Komponenten zu definieren. Die Schnittstellen werden als Interfaces in einem Klassendiagramm modelliert und schlussendlich in das Komponentendiagramm integriert.

Um die unterschiedlichen erlaubten Zugriffe zu modellieren, kann Vererbung genutzt werden: gemeinsame Schnittstellen werden in eine Basisklasse ausgelagert. Obwohl der Gateway bereits einen Großteil der Zugriffe regelt, sind wie in Kapitel 7.2 beschrieben zusätzliche Überprüfungen der Zugriffe von Nöten. Eine Visualisierung dieser verschiedenen Zugriffsrechte ist somit von Vorteil.

Ein Beispiel dafür ist die Beschwerdenschnittstelle des Beispielprojektes, welches sowohl eine Interne als auch eine Public Schnittstelle anbietet. Beide Schnittstellen haben gemeinsame Operationen, welche in eine Basisklasse ausgelagert wurden.

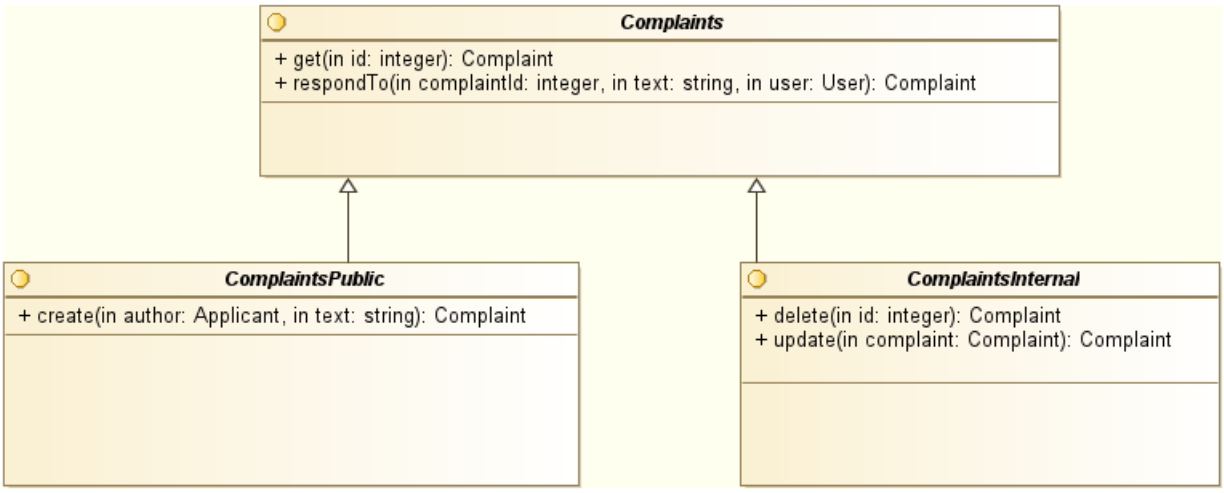

Abbildung 30: Erstellen der Verschiedenen Complaints Interfaces

Die verschiedenen Schnittstellen können nun in das Architekturkomponentendiagramm integriert werden. Dies wird für alle Komponenten durchgeführt und schließt die Architekturplanungsphase ab.

Nun kann mit der Implementierungsphase begonnen werden, welche sich nicht nur mit der Technologie, sondern auch der Programmstruktur beschäftigt. Die Implementierungsphase ist aus Gründen des Umfangs nicht mehr Teil dieses Prozesses.

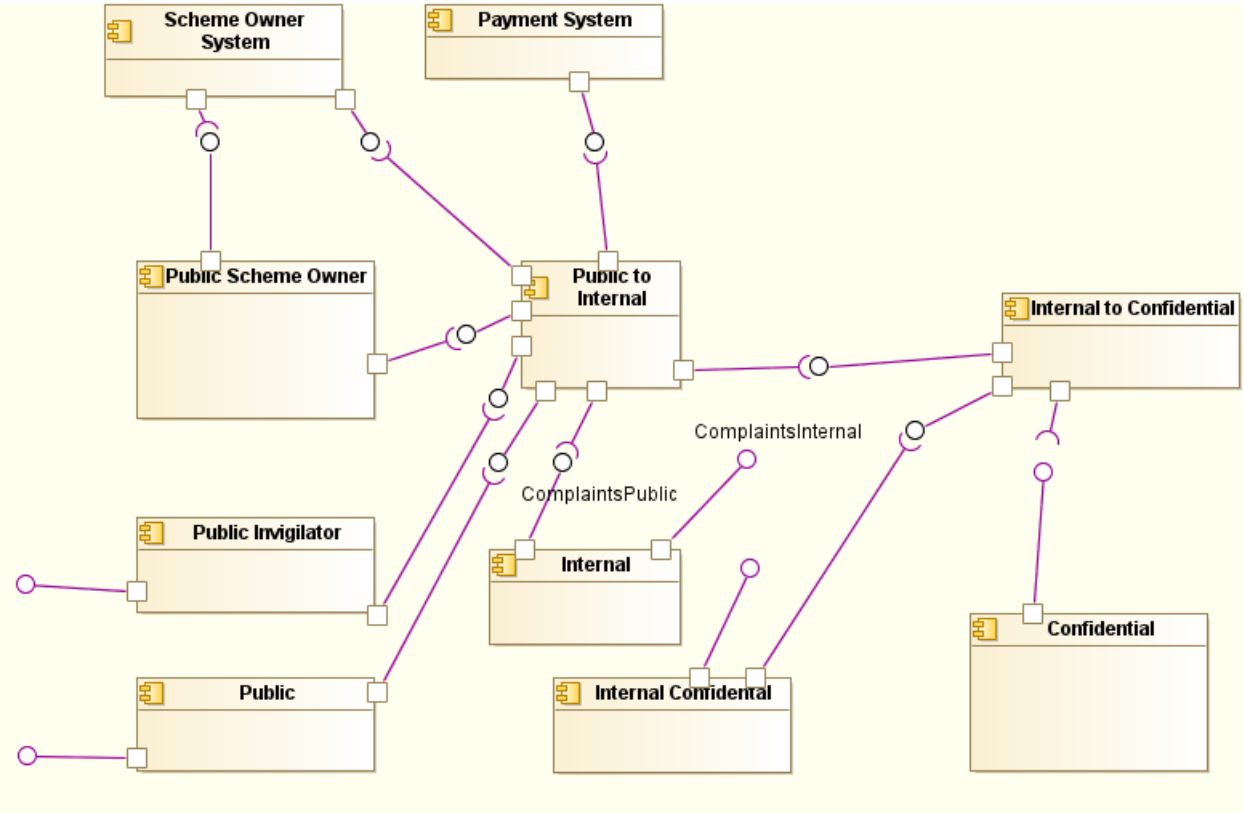

Abbildung 31: Verlinken der Complaints Schnittstelle

8 Zusammenfassung

In der Arbeit wurde anhand eines Beispielprojektes ein UML-basierter, allgemein anwendbarer Architekturprozess erstellt, mit welchem sich reproduzierbare Softwarearchitekturen erstellen lassen. Der Architekturprozess ordnet sich in die Planungsphase der Architektur ein und verwendet für die Rechtfertigung der Architekturentscheidungen einen kostenbasierten Ansatz.

Die initiale Frage - Wie kommt man von Anforderungen auf eine gute Architektur - konnte wegen der ungenauen Definition einer guten Architektur nicht genau beantwortet werden. Die Frage musste anders formuliert werden, nämlich wie folgt: Wie kommt man von Anforderungen auf eine angemessene Architektur. Eine angemessene Architektur ist eine Architektur, welche die Anforderungen des/der KundIn abdeckt, nicht Mehr und nicht Weniger.

Sollte nun diese Frage beantwortet werden, nämlich, ob die in der Planungsphase erstellte Architektur angemessen ist, musste die erstellte Architektur anhand von messbaren Werten

überprüft und mit den gestellten Anforderungen verglichen werden. Da in der Planungsphase noch keine Implementation des Systems vorlag, zeigte sich sich jedoch, das dies für die meisten nicht funktionalen Anforderungen äußerst schwierig ist.

Die zweite Herangehensweise für dieses Problem war, statt Messungen Schätzung bzw. Priorisierung zu fokussieren. Oliver Vogel schlägt hier folgende Formel vor: „Priorität = (Nutzen + Risiko + Wirkung) / 3“[33, S. 374]. Sind die nicht funktionalen Anforderungen priorisiert, kann nun aus einer Liste von bewerteten Komponenten/Architekturstile ausgewählt werden [20, S. 179] und die Architektur Stück für Stück zusammengebaut werden.

Diese Art der Architekturerstellung schien auf den ersten Blick eine gute Idee zu sein, entpuppte sich jedoch als sehr ungenau: Oft sind mehrere Anforderungen gleich wichtig; eine eindeutige Entscheidung ist somit in diesen Fällen schwierig. Außerdem sind auch die Variablen der Formel oft unklar: Was ist zB. das Risiko bzw. die Wirkung einer hohen Performance für den Usecase Anmelden? Eins, zwei oder drei? Ein weiteres Problem dieser Herangehensweise war die Bewertung der Komponenten bzw. Architekturstile: Die Architekturstile allein sind oft zu abstrakt, genaue Komponentenarchitekturen lassen sich damit schwer ableiten. Ein Versuch, alle möglichen Komponentenkombinationen abzudecken scheiterte am Umfang und der Vollständigkeit.

Da die meisten Architekturprozesse und -reviews wie ATAM die nicht funktionalen Anforderungen zur Entscheidungsfindung verwenden, ist dieses Ergebnis recht überraschend. Eine mögliche Erklärung hierfür könnte sein, dass ATAM Architekturreviews nicht nur in der Planungsphase durchgeführt werden können. Im Bezug auf CBAM lassen sich Parallelen erkennen, CBAM selbst aber bietet zwar ein Reviewframework aber keinen detaillierten Architekturerstellungsprozess.

Der erstellte Prozess begnügte sich daher mit schon messbaren, bekannten bzw. vergleichbaren Werten wie zB. den Kosten, welche unter Anderem durch die Verwendung von Risikoszenarien ermittelt wurden, zB. unerlaubter Zugriff auf Daten. Zusätzliche Komponenten wurden nur dann erstellt, wenn sich diese aufgrund der Kostenszenarien rechtfertigen ließen; dies führte zu einer kostenminimalen Architektur.

Außerdem wurde versucht, durch Analysen der nicht funktionalen Anforderungen der Architektur mögliche Risiken, Probleme und Parameter für die Entscheidungsfindung in der Implementationsphase zu generieren. Um die Fehlerkosten der Entscheidungen, welche auf den Analysen basierten, zu reduzieren wurden architekturrelevante Parameter schon so früh wie möglich ermittelt, nämlich in der Anforderungsphase.

Eine Architektur muss jedoch nicht nur für die Personen erstellt werden, welche sie tatsächlich umsetzen, sondern auch dem/der KundIn kommuniziert werden. Um dies zu erreichen können die kostenbasierten Entscheidungen und die UML Modelle herangezogen werden.

8.1 Vorteile des erstellten Architekturprozesses

Soll ein Architekturprozess für die Softwareentwicklung eingeführt werden, so muss dieser einen Vorteil bieten. Welche Vorteile zeichnen den in der Arbeit beschriebene Architekturprozess jedoch aus?

Die Vorteile ergeben sich im Prinzip aus den treibenden Anforderungen an den Prozess und das Projekt: Der Prozess soll nachvollzieh- und reproduzierbar sein, sodass verschiedene Personen mit den gleichen Anforderungen eine identische oder zumindest stark identische Architektur erstellen. Der Fokus liegt auf der frühen Planungs- und nicht der Implementationsphase, das Projekt selbst ist ein mittelgroßes, typisches Projekt und erfordert durch dessen Anforderungen eine hohe Datensicherheit.

Daraus ergeben sich folgende Vorteile:

- Gute Verständlichkeit
- Gutes Preis-Leistungsverhältnis

8.1.1 Gute Verständlichkeit

Der Prozess ist aufgrund der Verwendung von UML in einer Modellierungssprache dokumentiert, welche nicht nur über eine große Bekanntheit verfügt und viel Verwendung findet, sondern auch oft im Anforderungsprozess verwendet wird. Der Architekturprozess kann damit auf dem Anforderungsprozess aufbauen und Modelle weiterentwicklen anstatt diese in komplett neue Modellierungen zu überführen. Dies hilft nicht nur bei der Kommunikation mit dem/der KundIn sondern auch mit dem Anforderungsteam: Wenn sich zB. Anforderungen ändern, können diese an den gleichen Modellen geändert werden. Die erstellten Modelle dienen zugleich auch als Dokumentation des Projektes und helfen bei der Wartung.

Zusätzlich profitiert der Prozess von den generellen Eigenschaften einer visuellen Modellierungssprache: Sie bildet das komplexe auf eine einfachere Darstellung ab und verringert die Information auf die wirklich wichtigen Bereiche. Durch die visuelle Komponente ist das Modell schneller aufnehmbar und verständlicher als purer Prosatext.

Durch den Fokus auf Preis-Leistungsverhältnis ist es auch einfacher, die Probleme, Entscheidungen und Kosten dem/der KundIn zu kommunizieren, welcher wegen der fehlenden Vertrautheit mit dem Thema Softwarearchitektur oft die Entscheidungen und Notwendig dieses Bereiches anzweifelt [33, S. 8-9]. Da er mit Kostenfragen in der Regel vertraut ist, ergibt sich hier sogar eine Chance, bessere Anforderungen zu erlangen.

8.1.2 Gutes Preis-Leistungsverhältnis

Durch den Einsatz des Prozesses in der frühen Architekturplanungsphase lassen sich hier die meisten Fehlerkosten einsparen: Nach der Zehner-Regel der Fehlerkosten steigen die Fehlerkosten früher Fehler mit dem Projektfortschritt exponentiell an. Durch die frühe Einbeziehung

von Parametern, auf welche Architekturreviewszenarien aufbauen, lassen sich auch hier im Vorfeld Kosten reduzieren: Dies betrifft nicht nur die Ermittlung der Parameter selbst, sondern auch die Möglichkeit, triviale Probleme, welche erst bei einem Architekturreview offensichtlich werden können, durch diese Anforderungen schon im Vorfeld identifizieren und beheben zu können.

Eine weitere kostensparende Eigenschaft ist, dass architekturrelevante Parameter schon in der Anforderungsphase ermittelt werden und somit die Anzahl der KundInnenkontakte reduziert werden können.

Softwarearchitekturentscheidungen sind oft ein Trade-Off: Durch konkurrierende Qualitätsanforderungen, zB. Performance und Wartbarkeit, ist es in den seltensten Fällen möglich, eine Architektur zu erstellen, welche in allen Qualitätsanforderungen brilliert. Aus diesem Grund ist es notwendig, eine bestimmte Vorgehensweise zur Erstellung, Bewertung und Entscheidung von Architekturen zu pflegen. Der Prozess bietet in dieser Hinsicht ein kostenbasiertes Entscheidungs- und Analysemodell an, mit welcher diese Entscheidungen nachvollzieh- und nachrechenbar werden. Dadurch, dass weitere Systeme nur dann erstellt werden, wenn sie einen Kostenvorteil bergen, entsteht eine kosteneffiziente und preislich angemessene Architektur.

Nicht nur tatsächliche Kosten, sondern auch Risikokosten werden im Architekturprozess mit einbezogen. Der Fokus des Prozesses liegt auf Angriffs-, Aufalls- und Änderungsszenarien, welche als Resultat des Umfeldes des Beispielprojektes gesehen werden können.

Ein weiterer Kostenvorteil ist die Möglichkeit, Anschaffungen und Architekturentscheidungen aufzuschieben, da sich der Prozess nicht auf eine physische Architektur fest legt. Die wichtigen Grundsteine, die Schnittstellen und Kommunikationswege, sind bereits geregelt.

8.2 Limitierungen, Probleme und Nachteile des erstellten Architekturprozesses

Der Prozess ist zwar allgemein anwendbar, ist aber aufgrund der Herangehensweise für manche Gebiete schlechter geeignet. Die Probleme des sind vor allem in folgenden Bereiche auffindbar:

- Ausgreift- und Erprobtheit
- Vollständigkeit
- Universelle Eignung

8.2.1 Ausgreift- und Erprobtheit

Der Prozess wurde erst anhand eines einzigen Projektes erprobt und ging nicht über die Planungsphase der Architektur hinaus. Die Implementierungsphase und abschließende Überprü-

fung der implementierten Software wurde aus Zeit- und Umfangsgründen nicht durchgeführt. Dies führt nicht nur zu einer mangelnden Feedbackphase, in welcher Architekturprobleme offensichtlich hätten werden können, sondern auch zu einem zu starken Fokus auf die Domäne des Beispielprojekts.

Dies wird unter Anderem darin offensichtlich, dass der Prozess einen großen Wert auf Datensicherheit legt, welcher durch die dem Projekt zugrunde liegenden Anforderungen entstehen: Die Sicherheit und der richtige Umgang mit den KundInnendaten ist eine bindende Anforderung zur Erfüllung des ISO Standards für Personenzertifizierungsstellen, ohne welchen das System nicht betrieben werden könnte. [12]

Der Prozess ist somit noch unzureichend für den Einsatz in realen Projekten getestet. Vor Allem der Einsatz in unterschiedlichen Applikationsdomänen muss noch stärker geprüft werden.

8.2.2 Vollständigkeit

Die Probleme der Vollständigkeit des Prozesses sind unter Anderem eine Konsequenz der geringen Erprobtheit. Da es sehr viele unterschiedliche Risiken gibt und für unterschiedliche Projektfelder unterschiedliche Qualitäten verlangt werden, sind im Prozess nur ein Teil der möglichen Bereiche bearbeitet worden.

Außerdem behandelt der Prozess nur die Planungsphase und nicht die Architekturphase und beschäftigt sich somit weder mit der Strukturierung des Codes, noch gibt er Auskunft über die tatsächlichen eingesetzten, physischen Komponenten.

8.2.3 Universelle Eignung

Der Architekturprozess ist vor allem geeignet für Informationssysteme, deren Fokus auf folgenden Anforderungen liegt:

- Daten unterschiedlicher Vertraulichkeit müssen verarbeitet werden
- Datensicherheit ist wichtig
- Nicht jede AkteurIn kann auf alle Daten zugreifen
- Preis-Leistungsverhältnis ist wichtig für die Erstellung der Software

Liegt der Fokus jedoch auf anderen Bereichen, kann das Resultat der Analysen eine sehr simple Architektur sein, welche im Implementationsprozess stark angepasst werden müsste.

Ein Beispiel hierfür ist das Erstellen einer Architektur für ein sehr rechenintensives Systems, zB. für die Berechnung von Primzahlen. Hier ist es wichtig, die Berechnungen gut zu parallelisieren; der Architekturplanungsprozess würde aufgrund der Anforderungen eine einzige Softwarekomponente zur Berechnung der Primzahlen ermitteln, was zwar richtig wäre, aber wenig Hinweise zur Strukturierung der tatsächlichen physischen Systeme bieten würde.

Eine zweites Beispiel wäre ein Embedded System, welches eine starke Limitierung des Netzwerk- und Speicherdurchsatzes besitzt: Durch die Aufspaltung in mehrere Systeme, welche dann durch ein Netzwerk verbunden werden, würde die Zeit, welche zur Übermittlung der Daten aufgewendet würde, stark unter der ermittelten Architektur leiden. Auch eine Virtualisierung der Komponenten auf einem System würde wegen der Speicherlimitierungen nicht in Frage kommen. Die ermittelte Architektur müsste aufgrund der Hardwarelimitierungen stark überarbeitet werden.

8.3 Ausblick

Der Prozess wurde in der Planungsphase eines einzigen Beispielprojektes erprobt und schließt vor dem Beginn der Implementationsphase ab. Möglichkeiten zur Verbesserung des Prozesses lassen sich nicht nur durch die Erprobung weiterer, unterschiedlicher Projekte erlangen, sondern auch eine Erweiterung des Prozesses für die Implementationsphase ist denkbar.

Eine Möglichkeit für den Ablauf der Implementationsphase wäre folgender: Zuerst werden automatisierte Testfälle für die Qualitätsanforderungen erstellt. Die Komponentenarchitektur kann dann 1:1 auf die physische Architektur umgelegt werden, für jede Komponente wird ein physisches System eingesetzt. Die Funktionalität der Komponenten wird dann durch Prototypen umgesetzt. Anhand der Ergebnisse der Tests, wird regelmäßig überprüft, ob die geforderten Werte erreicht werden. Falls sie nicht erreicht werden, kann eine andere Architektur für die jeweilige Komponente anhand eines Prototypen überprüft werden.

Die Auswahl der Architektur, welche für die jeweilige Komponenten eingesetzt wird, obliegt den Entscheidungen der ArchitektInnen, sprich basiert auf deren Erfahrungswerten. Auch der Einsatz von Architekturreviewmethoden wie ATAM kann hier für die Entscheidungsfindung hilfreich sein. Ob die Architektur angemessen ist, kann überdies durch die bereits erstellten, automatisierten Tests überprüft werden.

Literaturverzeichnis

[1] *Bundesgesetz über den Schutz personenbezogener Daten*, 2015. http://www.ris.bka.gv.at/GeltendeFassung.wxe?Abfrage=Bundesnormen&Gesetzesnummer=10001597, Zugangsdatum: 17.04.2015.

[2] BILAL, H. und S. BLACK: *Computing Ripple Effect for Object Oriented Software*.

[3] COMMON VULNERABILITIES AND EXPOSURES: *CVE-2014-0160, Heartbleed bug*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160, Zugangsdatum: 18.4.2015.

[4] GARVIN, D. A.: *A framework for information systems architecture*. Sloan Management Review, (Fall 1984):25–43, 1984.

[5] HUMENBERGER, M., D. HARTERMANN und W. KUBINGER: *Towards a Model for Object-Oriented Design Measurement*. In: *Proceedings of the 5th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, S. 71–84. Springer, 2001.

[6] INSTITUTE OF ELECTRONICS ENGINEERS: *IEEE 1471*, 2000.

[7] INTERNATIONAL REQUIREMENTS ENGINEERING BOARD: *IREB Certified Professional for Requirements Engineering - Foundation Level*, 2015.

[8] INTERNATIONAL STANDARDS OFFICE: *ISO/IEC 10746*, 1996.

[9] INTERNATIONAL STANDARDS OFFICE: *ISO/IEC 9126*, 2001.

[10] INTERNATIONAL STANDARDS OFFICE: *ISO 42010*, 2011.

[11] INTERNATIONAL STANDARDS OFFICE: *ISO/IEC 25010*, 2011.

[12] INTERNATIONAL STANDARDS OFFICE: *ISO/IEC 17024*, 2012.

[13] INTERNATIONAL STANDARDS OFFICE: *ISO/IEC 27001*, 2013.

[14] JEROME H. SALZER AND MICHAEL D. SCHROEDER: *The Protection of Information in Computer Systems*, 1975. http://web.mit.edu/Saltzer/www/publications/protection/, Zugangsdatum: 12.3.2015.

[15] KAZMAN, R., M. KLEIN und P. CLEMENTS: *ATAM: Method for Architecture Evaluation*, 2000.

[16] KRUCHTEN, P.: *Architectural Blueprints - The 4+1 View Model of Software Architecture*, 1995.

[17] LARMAN, C.: *Applying UML AND Patterns*. Prentice Hall, New Jersey, USA, 2 Aufl., 2002.

[18] LYNCH, G. S.: *Single Point of Failure - The Essential Laws of Supply Chain Risk Management*. John Wiley & Sons, Inc., Hoboken, New Jersey, USA, 2009.

[19] MACIASZEK, L. A.: *Requirements Analysis and System Design*. Pearson Education Ltd, Edinburgh Gate, Harlow, England, 3 Aufl., 2007.

[20] MASAK, D.: *Der Architekturreview*. Springer, Heidelberg, Deutschland, 2 Aufl., 2010.

[21] MERRIAM-WEBSTER: *Process definition*. http://www.merriam-webster.com/dictionary/process, Zugangsdatum: 18.4.2015.

[22] MICROSOFT: *IIS remote code execution*. https://technet.microsoft.com/library/security/MS15-034, Zugangsdatum: 18.4.2015.

[23] O'HANLEY, R. und J. S. TILLER: *Information Security Management Handbook*. Taylor & Francis Group, Boca Raton, Florida, USA, 6 Aufl., 2014.

[24] OMG: *UML Specifications*, 2015. http://www.omg.org/spec/UML/, Zugangsdatum: 11.4.2015.

[25] PFEIFER, T. und R. SCHMITT: *Qualitätsmanagement: Strategien, Methoden, Techniken*. Hanser, München, Deutschland, 4 Aufl., 2010.

[26] POSCH, T., K. BIRKEN und M. GERDOM: *Basiswissen Softwarearchitektur*. dpunkt.verlag, Heidelberg, Deutschland, 2 Aufl., 2007.

[27] PRESSMAN, R. S.: *Software Engineering: A Practitioner's Approach*. The McGraw-Hill Companies, New York, USA, 7 Aufl., 2010.

[28] RUPP, C.: *Requirements-Engineering und -Management*. Hanser, München, Deutschland, 5 Aufl., 2009.

[29] RUPP, C. und S. QUEINS: *UML 2 glasklar*. Carl Hanser Verlag, München, Deutschland, 4 Aufl., 2012.

[30] SHORE, J. und S. WARDEN: *The Art of Agile Development*. O'Reilly, California, USA, 2 Aufl., 2008.

[31] STACHOWIAK, H.: *Allgemeine Modelltheorie*. Spring, New York, USA, 1 Aufl., 1973.

[32] STARKE, G.: *Effektive Software Architekturen*. Carl Hanser Verlag, München, Deutschland, 4 Aufl., 2009.

[33] VOGEL, O., I. ARNOLD, A. CHUGHTAI, E. IHLER, T. KEHRER, U. MEHLIG und U. ZDUN: *Software-Architektur*. Spektrum, Akademischer Verlag, Heidelberg, Deutschland, 2 Aufl., 2009.

Abbildungsverzeichnis

Abkürzungsverzeichnis

VPN Virtual Private Network

IREB Internation Requirements Engineering Board

CRUD Create Read Update Delete

ATAM Architecture Trade-off Analysis Method

CBAM Cost Benefit Analysis Method

ROI Return of investment

UML Unified Modeling Language

API Application Programming Interface

RM-ODP Reference Model for Open Distributed Processing